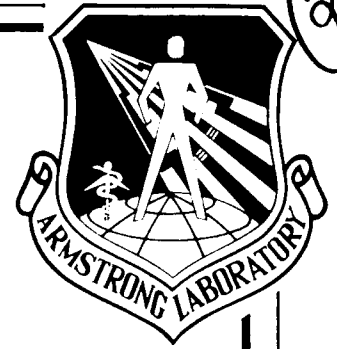


AD-A256 268

AL-TP-1992-0019



29

DTIC
ELECTE
OCT 16 1992
S C D

INFORMATION SYSTEM CONSTRAINT LANGUAGE (ISyCL) TECHNICAL REPORT

Louis P. Decker
Richard J. Mayer

KNOWLEDGE BASED SYSTEMS LABORATORY
DEPARTMENT OF INDUSTRIAL ENGINEERING
TEXAS A & M UNIVERSITY
COLLEGE STATION, TX 77843

HUMAN RESOURCES DIRECTORATE
LOGISTICS RESEARCH DIVISION

SEPTEMBER 1992

FINAL TECHNICAL PAPER FOR PERIOD JANUARY 1990 - MARCH 1991

Approved for public release; distribution is unlimited.

423810

92-27153



AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6573

ARMSTRONG

LABORATORY

NOTICES

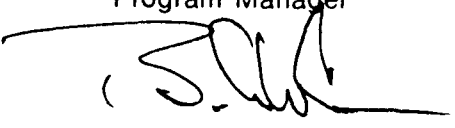
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation, or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this paper and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This paper has been reviewed and is approved for publication.



MICHAEL K. PAINTER, Capt, USAF
Program Manager



BERTRAM W. CREAM, Chief
Logistics Research Division

REPORT DOCUMENTATION PAGE			Form Approved OMB No 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1992	3. REPORT TYPE AND DATES COVERED Final - January 1990 - March 1991	
4. TITLE AND SUBTITLE Information System Constraint Language (ISyCL) Report Technical Report			5. FUNDING NUMBERS C - FQ7624-90-00010 PE - 63106F PR - 2940 TA - 01 WU - 15	
6. AUTHOR(S) Louis P. Decker Richard J. Mayer				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Knowledge Based System Laboratory Department of Industrial Engineering Texas A&M University College Station, TX 77843			8. PERFORMING ORGANIZATION REPORT NUMBER KBSL-89-1002	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Armstrong Laboratory Human Resources Directorate Logistics Research Division Wright-Patterson AFB, OH 45433-6573			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AL-TP-1992-0019	
11. SUPPLEMENTARY NOTES Armstrong Laboratory Technical Monitor: Capt Michael Painter (AL/HRGA), (513) 255-7775				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This paper describes a constraint language designed to serve as a Neutral Information Representation Scheme (NIRS) tying together model languages, procedural programming languages, database languages, transaction and process languages, as well as knowledge representation and reasoning control languages for information system specification. In one of its primary roles, the Information System Constraint Language (ISyCL) serves to augment the expressive power of existing systems engineering methods supported by a graphical languages. ISyCL is designed to be both powerful as a constraint language for completed information systems specification and easy to use by the various classes of users involved in information systems development. Expressive power is imparted to the language through the use of first-order predicate logic and set-theoretic constructs which provide the theoretical foundations of the language. Ease of use is promoted by providing layers within the language, ranging from natural language expressions to precise programming constructs, which shield users from unnecessary detail and complexity as dictated by their role in systems development.				
14. SUBJECT TERMS Constraint languages Engineering management information systems		Information management Information representation Information systems Programming languages		15. NUMBER OF PAGES 118
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	

Table of Contents

	<u>PAGE</u>
List of Figures.....	ix
List of Tables	x
Preface.....	xi
Summary.....	xi

Part One: Introduction and Overview

1. Introduction.....	1
Goals.....	3
Expressive Clarity	3
Enterprise Analysis	4
Method Integration.....	6
Currently Identified ISyCL Layers	8
ISyCL Characteristics.....	9
Readable	9
Object-Centered	10
Extensible.....	11
Intensions Versus Extensions.....	11
2. Rationale, Constructs, and Characteristics.....	13
Key Design Rationale	13
Infix Notation	13
Strong Type Checking	13

Accession For	
NTIS	ORAI
DTIC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability C	
Dist	Avail and
	Special

A-1

DTIC QUALITY INSPECTED 1

	<u>PAGE</u>
Case Sensitivity	14
Analyst Layer Constructs.....	14
Models	14
Quantifiers	16
If and Logical If.....	17
Set Builder	17
Operators.....	18
Systems Layer Constructs.....	18
Attributes.....	18
Sequences, Sets, and Bags	19
Sequences.....	19
Sets	20
Bags	21
Functions	21
Constraints (Functions with Triggers)	22
3. The IDEF1 Slice	24
Entity Classes	24
Owned Attribute Classes.....	25
Link Classes.....	26
Inherited Attribute Classes.....	27
Key Classes	28
Usage	28
4. IDEF1 Model with ISyCL Constraints	30
Model Element Representation.....	30

	<u>PAGE</u>
Constraints	32
Area Expert and Analyst Layers.....	33
Systems Layer.....	33
5. General Conclusions	36
Applicability.....	36
Effectiveness	36
Reliability.....	37
Verifiability.....	37
Future Directions	37
 Part Two: Language Reference	
6. Types.....	38
Type Declarations.....	39
ISyCL Types.....	40
Array	40
Bag	40
Boolean	41
Character.....	41
Number	41
Fixed.....	42
Float	42
Integer	43
Sequence.....	43
List.....	43
Ordered Set	44

	<u>PAGE</u>
String	45
Set	46
Symbol.....	46
Subtype and Subrange Definitions	47
Subtypes	47
Subranges.....	47
7. Expressions.....	48
Type Conversion	49
Operators	49
Arithmetic Operators.....	49
Assignment Operator	50
Boolean Operators	50
Logical If.....	50
Equivalence Operators.....	50
Function Composition.....	51
Membership	51
Relational Operators	51
Vectorization Operator.....	52
Linkage Operator.....	52
Precedence and Associativity.....	52
Quantifiers.....	52
Select.....	54
8. Statements.....	56
Common Structures	56

	<u>PAGE</u>
Conditional Branches	57
The <i>If</i> Statement.....	57
The <i>Case</i> Statement.....	57
The <i>Choose</i> Statement.....	58
Unconditional Branches	59
Iteration with <i>Loop</i>	61
9. Functions.....	64
Parameter Lists.....	66
Function Attributes	67
Macros	69
10. Classes	72
Objects	72
Entities	73
Behavior	75
11. Constraints.....	76
Triggers.....	76
Constraints	77
12. Metaclasses	79
Object Classes.....	79
Function Classes.....	81
Appendix A. ISyCL Grammar	86
Tokens	86
Productions.....	86

	<u>PAGE</u>
Lists of Tokens.....	87
Definitions	87
Types and Classes	87
Relational Operators	88
Boolean Expressions	88
Expressions.....	90
Declarations.....	90
Statements	90
Units.....	93
Meta Units	95
Appendix B. Revision Notes.....	96
Appendix C. References.....	100

List of Figures

	<u>PAGE</u>
Figure 1.1 IDSE Concept.....	2
Figure 1.2 IISEE Terminology Relationships	5
Figure 1.3 IISEE Terminology Relationships	6
Figure 2.1 IISEE Terminology Relationships	19
Figure 2.2 IISEE Terminology Relationships	23
Figure 3.1 IISEE Terminology Relationships	26
Figure 4.1 IISEE Terminology Relationships	30
Figure 6.1 IISEE Terminology Relationships	38
Figure 11.1 IISEE Terminology Relationships	78

List of Tables

	<u>PAGE</u>
Table 7.1 ISyCL Operators	49
Table 7.2 Operator Precedence and Associativity	53

Preface

This report describes the research accomplished at the Knowledge Based Systems Laboratory of the Department of Industrial Engineering at Texas A&M University. Funding for the Laboratory's research in Integrated Information System Development Methods and Tools has been provided by the Logistics Research Division of the Armstrong Laboratory (AL/HRG), Wright-Patterson Air Force Base, Ohio 45433, under the technical direction of USAF Captain Michael K. Painter, under subcontract through the NASA Research Institute for Computing and Information Systems (RICIS) Program at the University of Houston. The authors and the design team wish to acknowledge the technical insights and ideas provided by Captain Painter in the performance of this research as well as his assistance in the preparation of this report. Special thanks go to the ISyCL design team whose names are listed below:

Louis P. Decker
Keith A. Ackley
Richard J. Mayer, PhD
Christopher Menzel, PhD
Douglas D. Edwards, PhD
Joe! A. Toland
Thomas M. Blinn
Charles A. Bodenmiller

Summary

This paper describes a constraint language designed to serve as a Neutral Information Representation Scheme (NIRS) tying together model languages, procedural programming languages, database languages, transaction and process languages, as well as knowledge representation and reasoning control languages for information system specification. In one of its primary roles, the Information System Constraint Language (ISyCL) serves to augment the expressive power of existing systems engineering methods with graphical languages. ISyCL is designed to be both powerful as a constraint language for complete information systems specification and easy to use by the various classes of users involved in information systems development. Expressive power is imparted to the language through the use of first order predicate logic and set theoretic constructs which provide the theoretical foundations of the language. Ease of use is promoted by providing layers within the language, ranging from natural language expressions to precise programming constructs, which shield users from unnecessary detail and complexity as dictated by their role in systems development.

Part I

Introduction and Overview

Introduction

Since the earliest emergence of operating systems, database managers, and networking systems, it has been recognized that to build systems which are:

- flexible (can be economically modified),
- personal (can be tailored to individual needs),
- controllable,
- integrated (support reusable programs and shared data), and
- evolvable (less dependent on the underlying hardware),

one must incorporate into the system itself a form of the definition of the system which describes:

- its resources (data, hardware, programs),
- its interface to other systems (particularly the human systems it must support), and
- its users (their privileges, etc.),

among other attributes.

The community's understanding of the complexity of representing such a system definition has evolved from simple tables to logical schemas [ANSI 75], to "conceptual schemas" [ISO 82], to the realization that what is being represented is a complex knowledge base [PDES 88]. Within the Air Force Integrated Information Systems Evolution Environment (IISEE) program this problem has been studied from three different perspectives. The IISEE program is focused on the development of technology (theories, formalizations, frameworks, methods, automated tools and environments) for enabling (or improving the process of) the planning definition development and maintenance of evolutionary integrated information systems. Consequently, this program has studied the representational needs of this knowledge base from the point of view of the needs of engineering, manufacturing, and logistics Evolutionary Information Systems (EIS).

On the other hand, the definition of the concepts for a suite of automated tools and environments to support the pursuit of such engineering, manufacturing, and logis-

ties EISs has also required examination of languages and representation schemes for the knowledge base of these integrated systems (Figure 1.1).

Finally, one of the primary tasks of the IISEE program has been the definition of methods and mechanisms for integration of suites of systems engineering methods. As discussed in Mayer et al. [89], Wells and Mayer [88], this involves not only

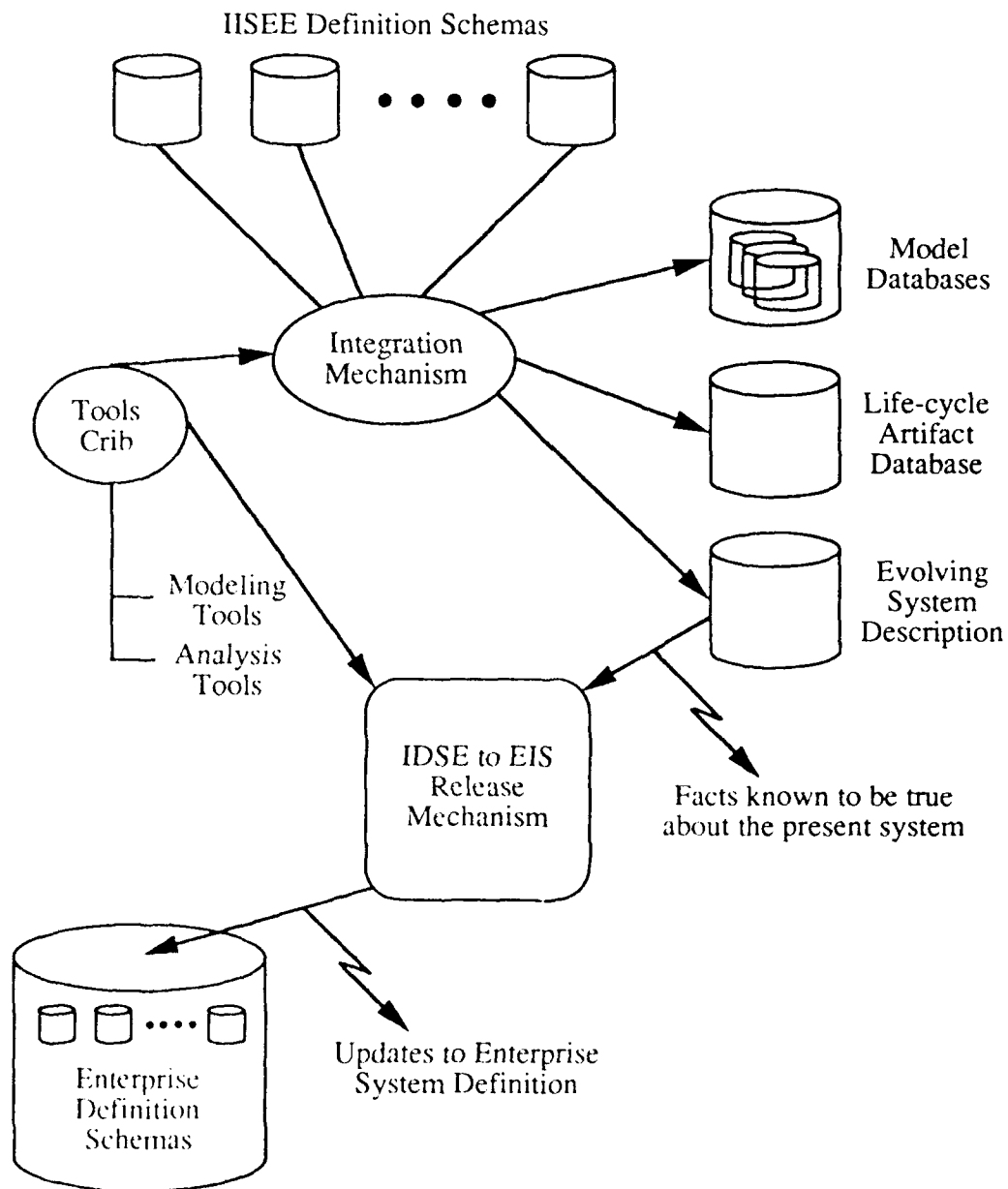


Figure 1.1 IDSE Concept

support for intermodel referencing, model data control, and intermodel data reuse but also interpretation of the meaning of one model's data and projection of that understanding into the perspective (as well as syntax) of another modeling language Wells and Mayer [88]. This form of integration requires a knowledge representation scheme on par with those used in natural language understanding, design synthesis, and automated program generation.

To accomplish its purpose, the language must be very rich. Such a system's definition language must be able to conveniently represent a complex range of information. For instance, the range of information may span from the existence of a data set on a particular disk drive, to the intent of a data flow modeler, to the business rules which govern the operation of a manufacturing enterprise and all its marketing, engineering, production, and logistics concerns. Thus, the language must support the following:

- Object Orientation,
- Relational Orientation,
- Persistent Storage,
- Process Transaction Specification, and
- First-Order Logic.

The task of designing a unified language which supports these paradigms (with both procedural and declarative styles of declaration) is certainly a challenge.

This document describes the Information Systems Constraint Language (ISyCL), which has been designed to meet the definitional and knowledge representation needs of the three perspectives described above. ISyCL, pronounced "icicle," is designed to support users from the area expert to the database designer. ISyCL is one language, but it has many layers which define subsets of ISyCL which are applicable to particular domains.

1.1 Goals

The following describes the primary goals of ISyCL.

1.1.1 Expressive Clarity

An important point not lost on the developers of ISyCL is that constraints are primarily for human consumption. What is a constraint worth if it cannot be verified by the experts in the field? Often it is easy to fall into the trap of writing statements so elegantly that no one can read them. When dealing with issues which weigh heavily not only economically, but also in human safety, constraints must be kept simple and concise.

The most readable and simplest form of constraint is the English sentence. Unfortunately, English statements can be misinterpreted if guidelines are not provided which reduce ambiguity and oversights. First-order logic restricts expression to a syntax which is thoroughly understood. Very few languages are understood as well as first-order logic. On the dark side, first-order logic is often difficult to read because of symbols used to make expressions concise.

ISyCL seeks to provide the clarity of first-order logic without forcing the user to accept the alien syntax. ISyCL "Englishizes" many of the logical constructs. For example, the existential quantifier (\exists) is replaced by *for_some*. This may seem trivial, but when perusing a page of constraints, the difference in readability can be enormous.

All in all, as a language, ISyCL must be able to answer the following positively:

- Is ISyCL applicable to the needs of the users?
- Is ISyCL effective in satisfying the needs of today's industry?
- Does the use of ISyCL produce reliable results?
- Are ISyCL constraints verifiable by the people capable of verification?

As this chapter progresses, each of these questions will be addressed in more detail. Also, please keep in mind that ISyCL is still in its infancy. It is a long road from inception to use in large-scale projects. Many forces will play a part in molding ISyCL into a mature language.

1.1.2 Enterprise Analysis

Where does one start in trying to model an enterprise? Figure 1.2 shows the key terminology relationships which are involved in the IISSE, of which ISyCL is a part. The right branch of the tree describes the computing environment which would support IISSE use. There are potentially many different development environments and under each environment there would be tools for the different methods and computer languages used to develop systems in that environment.

The left branch of the tree describes the modeling activity. There would be at least one framework (and possibly many) for relating the methodologies to the needs of the enterprise. Each enterprise would have a development procedure which provides the guidelines that describe how modeling is to be done. For instance, when planning a modeling project, the cost of the modeling must be weighed against the reduction in risk that results to find an affordable balance point. An enterprise would have guidelines for acceptable levels of risk and so forth.

Many methodologies are used in modeling an enterprise. It is not reasonable to expect one methodology to be able to model all aspects of an enterprise. There are information, data, activity, process description, and many other methodologies. Each

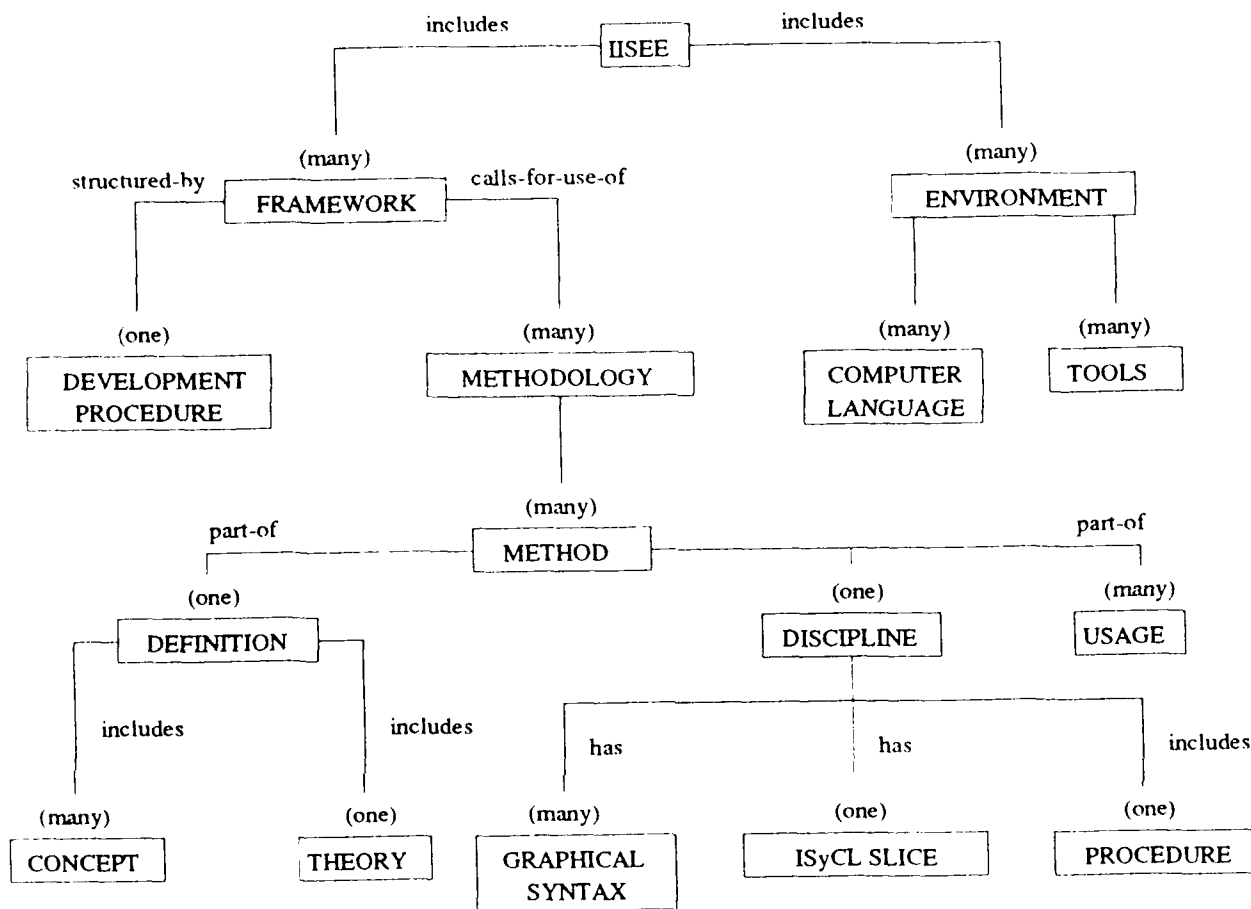


Figure 1.2 ISEE Terminology Relationships

methodology has many methods. For example, IDEF0 is a method for activity modeling.

Each method has a definition, discipline, and many uses. The definition contains the concepts involved and the theory behind the method. The discipline includes the syntax of the method and the procedure by which the method is applied. Many methods have multiple syntaxes which have either evolved over time, or are used for different aspects of the modeling.

A method's procedure is very important to the successful application of the method. The reader is directed to the procedures of IDEF0 [SofTech 81], IDEF1 [Mayer 87], and IDEF1x [DACOM 85], since there is no need to repeat here the information in those excellent texts.

1.1.3 Method Integration

ISyCL is more useful if data are collected properly. The resulting model(s) will be more reliable, and the integration of models from different methods becomes more reasonable. Since ISyCL is meant to act as the neutral representation format for allowing integration of models, ISyCL's syntax seeks to enforce consistent naming and meaning.

Figure 1.3 shows the process which would occur when a model is entered using one method and then translated to a different method. Starting at the left, the modeler enters the model (using an automated tool) and the constraints which cannot be expressed in that method. He or she uses the method's syntax (usually graphical) to enter the model. ISyCL constraints would be expressed using the slice of ISyCL (the subset of ISyCL specific to a method) for that particular method. For instance, the structures used to describe constraints on an IDEF3 process description would be different than those used to add constraints to an IDEF1x data model.

The translation of the graphical model to an ISyCL representation is accomplished by the modeling tool using meta layer definitions which had been defined earlier. Very few people will ever see these definitions, much less write them. There should be a standard set for each method. The ISyCL representation of the model, plus any constraints is considered to be the neutral representation of the model.

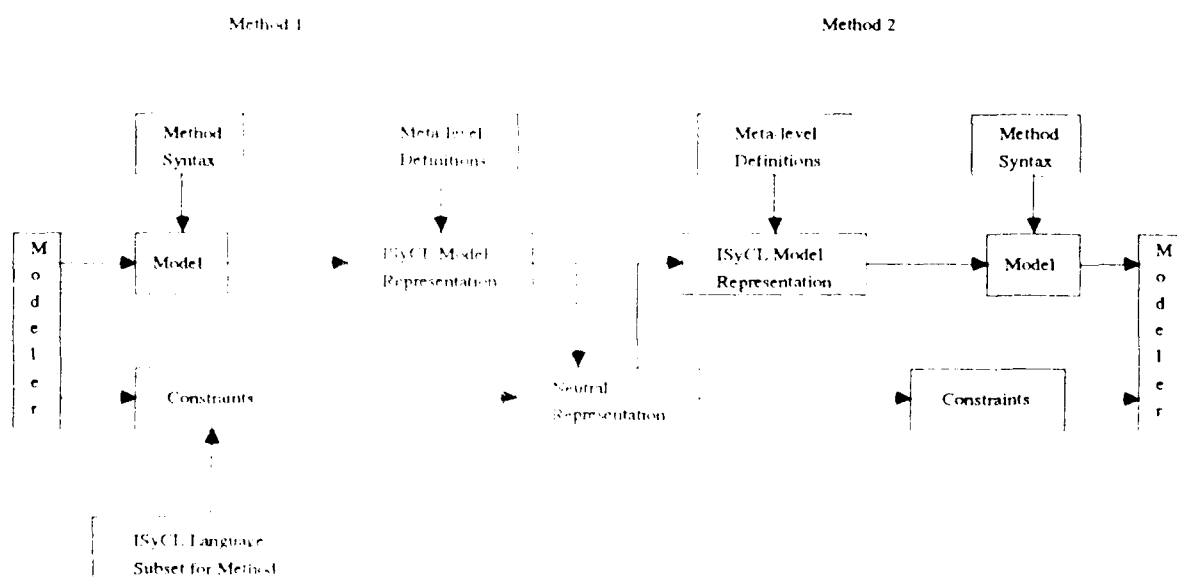


Figure 1.3 Model Translation

Being able to project information presented in the syntax of one method into the syntax of another is not only desirable, but will prove indispensable for highly complex systems requiring multiple discipline involvement. To accomplish this, an expert system would be required which uses the translation techniques discussed in Mayer et al. [89]. The translation would map between the meta layer definitions of the two methods, probably using tags and other user-supplied aids. For example, an IDEF1 entity class might map to an IDEF1x entity, but the modeler would have to designate the primary key of the IDEF1x entity, since IDEF1 does not differentiate between key classes.

Each method would have translation rules which are used to generate a textual representation of the information in a model. Once the information is in a textual format it can be used: (1) to evolve the model of the enterprise (the Evolving System Description, or ESD), (2) by model translation facilities for generating views using other methods, or (3) as input to other modeling tools. In order to generate this textual representation, a language is needed which can express the information contained in models from any given method. ISyCL is the first implementation of such a language.

One of the most difficult aspects of information system design is identifying what information is really being tracked. Different methods present different views of the information (e.g., data, activities, etc.). On the other hand, there is overlapping information between models in different methods. There needs to be a way of understanding what information is being maintained in order to reduce inconsistencies and missing information between models. Since most methods have been developed independently of one another, the underlying foundations of the methods do not readily facilitate integration of models in different methods. This is not to put the blame on the developers of the methods. It is a very difficult task to create a set of *useful* methods which span the development life cycle. In fact, it has yet to be done. The current methods provide the means to tackle large projects by concentrating on specific areas. When producing a model, other types of models can be used as reference as long as the modeler understands the intentions of the other modelers. There is no “plug & play” integration between models in different methods. Each link between the models must be added by the modeler after careful consideration.

Given that a modeler can identify the links between models, how can those links be maintained? This is where the Integrated Design Support Environment (IDSE) can be of assistance. The IDSE allows tools to be used which are not customized for the environment. Thus, one tool will not necessarily know of the existence of other tools. Consequently, the links cannot be maintained at the tool level. The IDSE will have to maintain these links given the ISyCL descriptions of the models. These links can just be looked at as special constraints which span models. When a modification is made to a model containing links to other models, these linkage constraints must be checked to determine if other models need to be modified. Thus, linkage constraints are not necessarily information constraints as much as integration constraints. They constrain the IDSE, whereas the information constraints constrain the actual infor-

mation objects in the enterprise's information systems. An example integration constraint could look like:

`Info_Model::Entity_Class:Engineer <==> Act_Model::Concept:Engineer`

where, *Info_Model* is an IDEF1 information model and *Act_Model* is an IDEF0 activity model. The constraint states that the entity class, *Engineer*, is linked to the concept, *Engineer*. If either is deleted from its respective model, the other would be marked questionable. If an arrow head appears on only one side of the linkage operator, the arrow points from independent to dependent. If a dependent is deleted, it does not affect the independent. On the other hand, if the independent is deleted, then the dependent is marked questionable.

Let's look at the scenario by which a linkage constraint would be added to a system. First, the two models would be developed using the appropriate tools. During the creation of the models, the modeler would identify the linkage constraint. Unless more automated support was available, the modeler would enter the constraint like other ISyCL constraints using a text editor. Upon completion and release of the models to the ESD, the IDSE would parse those linkage constraints and identify the links. If one or the other of the models is modified in the future, those links will be checked.

This integration mechanism is simplistic, yet useful. It prompts the user for maintenance instead of trying to automatically determine what actions need to be taken. In other words, integration is facilitated but not automatic.

The expert translation system would also need to decide what portion of the neutral representation can be described in the method and what will remain as ISyCL constraints. Naturally, as much should be expressed in the method's syntax as possible. Next, a tool would take the ISyCL representation of the model and display it using the method's syntax.

1.2 Currently Identified ISyCL Layers

There are currently four defined ISyCL layers:

- Area Expert Layer,
- Analyst Layer,
- Information Systems Design (Systems) Layer,
- Method Formalization (Meta) Layer.

The layers represent the view of the information system taken by different people in an enterprise. The area expert looks only far enough to make sure that the information being kept is consistent with his or her expectations. The analyst looks deeper to see how information from different experts is related. Finally, the information system

designer takes the information obtained by the analyst and implements the actual information system.

Methods with graphical languages are used to describe or model information about (or contained in) a particular domain of interest. For instance, IDEF0 could be used to describe the activities involved in producing an automobile. Automobile manufacturing would be the domain and IDEF0 would be a suitable method. The analyst layer of ISyCL is used to describe constraints on models which have been created using the various methods.

Another goal of ISyCL is to facilitate the transition of models into information system implementations. Naturally, when implementing an information system, more attention must be paid to programming issues. Thus, in this layer ISyCL provides a more "programming language" type of syntax. This layer is called the information systems design (systems) layer.

Besides specifying constraints on model elements in a method diagram, ISyCL must also be able to describe the model elements themselves. This requires defining an information model of the methodology (or metamodel), and then using the method formalization (meta) layer of ISyCL to define those model elements. The meta layer definitions are then used at other layers to describe constraints and design information systems.

The method formalization layer is different from other layers in that it is used to define "slices" of ISyCL which are used in conjunction with different methods. For instance, support for a process description method like IDEF3 requires different constructs than support for an information modeling method like IDEF1.

1.3 ISyCL Characteristics

Care has been taken to keep ISyCL's syntax as clear and familiar as possible. For instance, much of the Structured Query Language (SQL) standard has been incorporated into ISyCL's syntax. On the other hand, ISyCL provides structures like the set-builder notation (e.g., {x : integer | x < 100}) from logic for those who prefer a less procedural syntax.

1.3.1 Readable

Since one use of ISyCL is to specify textual constraints on models developed using systems engineering methods with graphical languages, ISyCL's syntax must enforce readability. ISyCL statements may be as simple as:

```
for_all e of Employee
  (US_citizen?(e))
```

which reads, "For all members of class *Employee*, check the member's citizenship." If all employees are U.S. citizens, then this statement is true, else it is false. Statements may also be as complex as:

```
after init of entity_class:Employee e ()
  [unless (US_citizen?(e))
    raise(not_US_citizen, e);]
```

which reads, "After initialization of an entity of entity class, *Employee*, called *e*, unless *e* refers to a U.S. citizen, raise the exception, *not_US_citizen* (passing *e*)." The exception handler would take appropriate actions if the employee was not a U.S. citizen.

The first example is just a statement that all employees must be U.S. citizens, whereas the second example declares when to check instances and what to do if one refers to a non-U.S. citizen. By necessity, the second example is more complex. An analyst might write the first statement, while a systems designer might write the second. Even still, it is important that statements remain as readable as possible no matter what level of detail is being expressed.

1.3.2 Object-Centered

ISyCL provides an object-centered approach to model elements and provides facilities for attaching constraints to the model elements. Model elements are either object classes or function classes. For example, IDEF1 entity classes are object classes and IDEF1 link classes are function classes.

Instances are elements which conform to the requirements of a class. In other words, instances are members of the class. Some instances are persistent (stored in a database), others are dynamic (not persistent across sessions), while others are instances of functions (executable code).

ISyCL provides facilities for defining new classes. Classes have attributes which describe the instances of that class. Thus, a "car" class would have attributes like "color."

ISyCL *objects* are dynamic data structures used to facilitate programming through the object-oriented paradigm. ISyCL and the object-oriented design method, IDEF4, have a special relationship in that ISyCL can be used to add constraints to IDEF4 models and IDEF4 can be used to model designs to be implemented in ISyCL.

ISyCL *entities* are "information system objects," which represent information about real or abstract objects. *Entities* are persistent until deleted from the information system. For example, information about an employee resides in the information system until that employee departs and can be forgotten.

One of the most often used constructs in ISyCL is the *trigger*. Triggers can be applied before, during, or after functions. All user-defined objects and entities have standard functions for instantiation, modification, and deletion, called *init*, *mod*, and *del*, respectively. These are the most often constrained functions.

1.3.3 Extensible

ISyCL must be able to not only allow constraints to be added to models developed using any graphical method, but also to represent the information contained in the models. In order to support methods which have not even been developed yet, ISyCL must be extensible. ISyCL is extensible through the method definition (meta) layer and what are called "slices." Each method would have a slice of ISyCL which describes the model element types for that method. These slices are defined using ISyCL's meta layer. Slices would be defined by appointed experts in a given method.

1.4 Intensions Versus Extensions

We need to take a moment to discuss intensions and extensions (yes, our intention was to spell "intensions" with an "s"). An intension describes "all possible" members of a class, while an extension is the set of currently known members of a class.

It is very important to be wary of intensions and extensions. For example, say a person were new to earth and he or she had met only people with brown hair. By extension, the person would define that people have brown hair. If he or she could see the master plan of the human being, the codomain (the possible values) of hair color could be identified. The master plan for human beings is intensional in that it describes all possible human beings. ISyCL supports both intensional and extensional definitions.

Another example is the specification of a constraint upon employees of a company. The company does not allow two employees to have the vault key at the same time. This constraint could be written: "Given any two distinct employees, both cannot have the vault key." That constraint upon an IDEF1 entity class *Employee*, could be written in ISyCL as:

```
for_all x,y of entity_class:Employee where (x <> y)
  (not (key?(x) and key?(y)))
```

This statement is intensional because it states that for all *possible* employees no two can have the vault key.

On the other hand, we could say that no two employees from a set can have the vault key. This could be written:

```
for_all x,y in {e, f, g} where (x <> y)
  (not (key?(x) and key?(y)))
```

where, *in* specifies that we mean the employees (referred to by *e*, *f*, and *g*) in the given set. In this case the constraint covers the given extension of *Employee*.

It is important to note what types of constraints are usually intensional versus which are extensional. Most model elements in a systems engineering method are intensionally defined, whereas metrics are usually extensionally defined. Business rules are usually extensionally defined and logical constraints are usually intensionally defined, hence the design problem of making the translation between the two.

Intensional constraints are very powerful, but they may lead the designer into possible misassumptions. It is easy to make broad, sweeping generalizations which overlook a subset of the members. Also, one might not take into account future evolution of the enterprise.

As the language is defined, the usage of intensionality and extensionality will be described in more detail. For a formal description of intensional structures in IDEF1, see Menzel and Mayer [89].

Rationale, Constructs, and Characteristics

ISyCL was initially conceived only as an IDEF1 extension to assist with metamodel specification. During its development, it was determined that it needed most of the constructs necessary for the Neutral Information Representation Scheme (NIRS), the language which would serve as the glue between the cells in the Zachman Framework [Zachman 86]. Such glue must tie together model languages, procedural programming languages, database languages and, transaction and process languages as well as knowledge representation and reasoning control languages. Thus, though analysts may never need many of the programming constructs which are part of ISyCL, the constructs are nevertheless there. The layers shield users from the constructs which are not necessary for them to complete their tasks.

2.1 Key Design Rationale

Even at the systems layer, ISyCL's syntax enforces readability. Thus, even though attention to implementation is important in an information system design, computer system details such as memory locations and the like are purposefully hidden by ISyCL. It is up to an implementation of ISyCL to translate ISyCL descriptions to machine-level code.

2.1.1 Infix Notation

Infix notation was also chosen for ISyCL to provide readability. It may seem to some that there could have been no other choice, but such was not the case. The design group actually switched to parenthesized prefix notation at one point due to the elegance of such notation, but it was decided to place readability by laymen before elegance.

2.1.2 Strong Type Checking

ISyCL's syntax forces declaration of types to check intention as well as to ease compiler implementation. As will become evident, ISyCL tries to maintain the clarity of first-order logic while still providing programming constructs which permit efficient applications.

2.1.3 Case Sensitivity

Another critical choice in the design of ISyCL concerned case sensitivity. In many situations, case sensitivity can lead to errors which are difficult to find. It also forces the user to remember (or look up) the proper case of symbols. On the other hand, case sensitivity allows greater flexibility and semantic content of symbols. Most important, case sensitivity allows functions written in other case-sensitive languages to be called without name mapping. Since ISyCL is a special purpose language, it would be desirable to interface easily with other languages.

A compromise has been made which allows case sensitivity when necessary. ISyCL symbols are not case sensitive unless a “%” is placed in front of them. In other words, the processor converts all symbols to uppercase unless there is a “%” in front of them.

Even though ISyCL is not case sensitive, for readability it is suggested that names of persistent object classes be capitalized so that it is obvious to the reader that these are special classes. It is also suggested that names of variables whose values have units be in upper case.

2.2 Analyst Layer Constructs

In this section we will describe the primary constructs that would appear in the analyst layer subset for most analysis, requirements, and design methods. Many of these constructs find their origins in first-order logic. Other constructs, not described here, are specific to certain methods. For instance, temporal relationships lie within the domain of process flow modeling, but have no place in information modeling. Discussion of constructs which are specific to certain methods will be deferred to later sections.

2.2.1 Models

Since ISyCL will be used to add constraints to models in graphical languages and to translate between methods, it must be able to differentiate between a class based on one method's meta layer constructs and another class with the same name which is based on a different method's meta layer definitions. For example, there may be an IDEF1 entity class named *Employee* and an Entity Relation (ER) entity set named *Employee*.

To accomplish this, ISyCL uses a construct which is similar to “packages” in Common Lisp. At the top of each file of ISyCL definitions, an attribute line will describe (among other information) which model is the default model for that file's definitions. For instance,

```
-- Model: KBSL
```

would appear at the top of a file of constraints for the “KBSL” model.

Models can only be defined using one method.¹ The “KBSL” model would have a definition similar to:

```
model KBSL (IDEF1);
```

All classes used in such a model would either be based on built-in metaclasses or metaclasses defined for IDEF1.

Of course, it may also be desirable to have two classes with the same name within the same model which are based on different metaclasses. In an ENALIM model it might be desirable to have an *Employee* NOLOT (Non-Lexical Object Type) and an *Employee* LOT (Lexical Object Type). Thus, given the statement,

```
for_all e of Employee
  (salary(e) > 0)
```

it would not be possible to tell which *Employee* class was being referenced. To differentiate between metaclasses, ISyCL allows a prefix to be placed before the class. The prefix is the name of the metaclass or a defined abbreviation for the metaclass name. We might have written the previous constraint like:

```
for_all e of LOT:Employee
  (salary(e) > 0)
```

which designates that we mean the *Employee* LOT.

It may also be possible that different methods will have metaclasses of the same name. Thus, if in the middle of a file of constraints on an ENALIM model, it was necessary to reference an IDEF1 *Employee* entity class from the “KBSL” model, one would reference it by:

```
for_all e of KBSL::entity_class:Employee
  (salary(e) > 0)
```

It should be clear that constraints should be kept together in a file which has a default of the model being constrained. Otherwise, naming can get messy.

¹ Note that “method” always refers to “modeling method” since ISyCL does not use the term “method” to refer to functions associated with a class.

2.2.2 Quantifiers

Nearly every constraint starts with a quantifier. There are two types of quantifiers, universal and existential. Quantifiers test a relation across the members of a class or set. Quantifiers act as predicates, which means they are expressions which return either true or false. Universal quantifiers test members until one is found which fails the condition. If a member fails, the universal quantifier returns false. If none fail, the universal quantifier returns true. An example of a universally quantified expression is:

```
for_all e of entity_class:Employee
  (age(e) >= 16)
```

which checks to see whether all employees are at least sixteen. The symbol following the *for_all* represents a member of the set or class being checked. It is possible to check all possible pairs of members using:

```
for_all x,y of entity_class:Employee
  (married?(x,y))
```

which checks for possible nepotism.

If a set is to be checked, then *of* would be replaced by *in*. Since marriage is a reflexive relationship, there is no need to check if "John" is married to "Sue" if we already know that "Sue" is not married to "John." Thus, it would be more efficient to check only the unique pairings. This is done like:

```
for_all (x,y) in pairs(entity_class:Employee)
  (married?(x,y))
```

Pairs generates the set of pairs in which members are only paired with one another once. The special (x,y) notation causes the first member of each pair to be bound to x and the second to y .

In the examples above, the IDEF1 *entity_class* metaclass is used to describe the class of the class *Employee*.

After the type specification comes the condition to be checked. Conditions are always placed within parentheses. Quantifiers can be nested as in:

```
for_all d of entity_class:Department
  (for_some e of entity_class:Employee
    (dept(e) = d))
```

which determines if all departments have at least one employee. Note the existential quantifier *for_some*. *For_some* is like *for_all* other than that it returns true if at least one member satisfies the condition.

We did skip over one option. Say we wanted to check all employees of the “Engineering” department to make sure they are all classified as engineers. We would use:

```
for_all e of entity_class:Employee | (name@dept(e) = "Engineering")
    (engineer?(e))
```

which checks all employees in the engineering department to see if they are all engineers. The “|” is read as “such that” or “where.” If the bar seems too concise, *where* can be used instead. *Name@dept(e)* is the same as *name(dept(e))*. The “@” is the function composition operator.

2.2.3 If and Logical If

There are other ways we could have checked the *Engineering* department employees instead of using *where*. *If* and *logical if* could also have been used. *If* is the standard programming *if*, where if the condition fails, *false* is returned. Conversely, *logical if* returns *true* if the condition is *false*.

For instance,

```
for_all e of entity_class:Employee
    (if (name@dept(e) = "Engineering")
        engineer?(e);
        else true;)
```

is a somewhat contrived form. If an employee is a member of *Engineering*, the employee is checked, else true is returned which keeps the iteration going. This form is difficult, because *if* returns *false* if its condition fails and there is no *else* form.

Since this type of structure is common in logic, ISyCL has the *logical if* which returns *true* if its condition is false, like:

```
for_all e of entity_class:Employee
    (name@dept(e) = "Engineering" -> engineer?(e))
```

This form accomplishes the same feat as the others. In fact, people with logic backgrounds would probably write this instead of using *where* (!). However, it is a little more confusing for laymen, especially programmers.

2.2.4 Set Builder

Sets play an important role in many constraints, and ISyCL provides a powerful construct for generating sets. The set builder form collects a set of entities which satisfy a condition. For instance,

```
{e of entity_class:Employee | name@manager_of(e) = "Bob"}
```

collects the set of employees whose manager is named "Bob." The syntax is very similar to the quantifiers we discussed earlier. There is almost always a *where* clause in a set builder form.

2.2.5 Operators

ISyCL includes all the common boolean operators (*and*, *or*, *not*) plus *xor* and *iff*. *Iff* is the same as boolean equivalence, and is used often in logic. ISyCL also includes the standard mathematical operators "+", "-", "*", "/", and "^", where "^" is the exponent operator. Note that ISyCL does not have a modulus operator. There are two reasons for this. First, ISyCL is not a general-purpose language and finding a modulus is not commonly needed when writing constraints on information systems. Second, there would be a name conflict with the *mod* function which is called to modify the value of attributes (see page 11).

Since sets are important, ISyCL also provides set operators. *Union* creates the set containing all members of two sets. *Inter* returns the intersection, or common members, of two sets.

2.3 Systems Layer Constructs

The following constructs, though some are available in the analyst layer, are described from the perspective of the systems layer of ISyCL. When used by an analyst, the implementation details described below would not be of importance.

2.3.1 Attributes

Though attributes are commonly referenced at the analyst layer, attribute values have no real use until the information system is implemented.

ISyCL models attributes as *functions* which map objects or entities to attribute values either stored or derived. This is a major difference between ISyCL and languages like the Structured Query Language (SQL) and C++. In SQL, attribute values are referenced by treating the attribute like a slot in a C structure (e.g., *person.hair_color*). This method ensures that the reader differentiates between stored values and derived values. Actually, the reader need not care whether the value is stored or derived; it is just an attribute value.

Consequently, in ISyCL attribute values are referenced by applying the attribute to the instance (e.g., *hair_color(person)*). To reduce confusion, the use of periods for function application (e.g., *hair_color.person*) is not allowed (even though it may have been desirable in some cases).

2.3.2 Sequences, Sets, and Bags

ISyCL provides many high level data structures which are useful when describing constraints on information systems. By providing these data structures as standard types, analysts and information system designers can rely upon their existence. Many languages force programmers to “roll their own” high level data structures. Since many users of ISyCL will not be programmers, it is preferable to build these data structures into the language.

The standard functions for manipulating these higher level data structures are yet to be defined. Figure 2.1 shows the relationship between lists, bags, sets, and ordered sets.

2.3.2.1 Sequences

Sequences are structures used to hold groups of ordered elements. There are four different types of sequences:

- Vectors,
- Lists,
- Ordered Sets,
- Strings.

There are many standard functions for manipulating sequences. These functions will accept any subtype of *sequence*. The types of sequences are differentiated by their storage requirements, element types, and allowance of duplicate members.

Vectors are one-dimensional arrays. Elements of a vector are referenced by specifying the number of the element in square brackets after the name of the variable which

	No Order	Order
No Duplicates	Sets	Ordered Sets
Duplicates	Bags	Lists

Figure 2.1 Sequences, Sets, and Bags

refers to the vector. For instance, $v[8]$ would refer to the eighth element of the vector which is the value of v .

Strings are similar to vectors, but strings have only character elements. Also, to reduce the complexity of using strings, the quantity of storage necessary to hold the string is automatically adjusted by the ISyCL processor. Thus, it is not necessary to declare the maximum length of a string variable.

Lists are ordered sets of elements. Lists allow duplicate elements and can be arbitrarily long. Lists are commonly used data structures in programming, but less frequently used in logic. Ordered sets are like lists, only they do not allow duplicate elements.

The following are example declaration statements for sequences:

```
v :      vector(5, character), init_value make_vector(5, character,
                                                    (init_elements ('H, 'a, 'r, 'r, 'y')));

s :      string(11), init_value "Harry";

l :      list_of(character), init_value list('H, 'a, 'r, 'r, 'y);

o :      ordered_set_of(character), init_value ordered_set('H, 'a, 'r, 'r, 'y) ⇒ ERROR!
```

Each of these examples seeks to group the five characters in "Harry." The vector has the advantage of being able to efficiently access random characters in the name, but it is not flexible. For instance, v could not be assigned the characters in "Harry Jones" because it can only contain five elements.

The string declaration is the most appropriate. The optional parameter declares that enough room should be reserved for eleven characters. If this value had not been provided, s could still have been assigned "Harry Jones," but the assignment would be less efficient since the system might have to allocate more space on the fly.

The list declaration is flexible in that as many characters can be added as necessary, but it is unlikely that one would want to store a name in a list structure. The ordered set declaration is illegal because ordered sets cannot contain duplicate elements.

2.3.2.2 Sets

Sets are one of the most widely used structures in ISyCL. Sets behave like sets in logic (i.e., no order, no duplicate occurrences, etc.). Many operations can be performed on sets of entities or objects. For instance, the *for_all* construct shown earlier will check all members of a set to see if they satisfy a condition.

Sets of entities form the extensions of intensionally defined classes. For example, a company may keep track of how long its managers have been with the company.

Thus, the entity class, *Manager*, would have the attribute, *time_with_company*. A specific manager is an entity, and the amount of time that manager has been with the company is an attribute value. The extension of the entity class, *Manager*, is the set of all current managers.

One could construct the set of managers who have been with the company greater than twenty years by writing either:

```
{m of entity_class:Manager | time_with_company(m) >= 20}
```

or,

```
select * from entity_class:Manager
  where (time_with_company >= 20)
no duplicates;
```

These two forms return the same result. It is up to the user to determine which feels more comfortable. The *select* expression is a powerful construct in ISyCL and can be used to generate sets, ordered sets, lists, and bags. The syntax of the *select* expression is kept as close to the SQL statement as possible.

2.3.2.3 Bags

Bags are unordered collections of elements. Bags can contain duplicate elements. The bag of managers with better than twenty years seniority would be selected by:

```
select * from entity_class:Manager
  where (time_with_company >= 20);
```

2.3.3 Functions

ISyCL functions return values of a specified type or no values at all. Functions are similar in purpose to functions in other languages. The value of the last statement executed in a function is the returned value unless a return statement is used explicitly. An ISyCL function which returns the set of employees whose age is greater than any member of department *E21* is:

```
function Old_Timers () : set_of(Employee)
  "Find all employees whose age is greater than any member of department E21."
  [select * from Employee
    where (age >= (select max(age) from Employee
                  where dept = "E21");))
  no duplicates;]
```

The first line declares *Old_Timers* to be a function of zero arguments which returns a set of *Employees*. The parameter list is required, but if there is no codomain specified, it is assumed that the function does not return any values. The documentation string (*doc_string*) which follows is supposed to describe the purpose of the

function and explain any important points about the implementation from a user's perspective.

The body of the function is composed of a *select* statement which in turn uses a nested *select* in its *where* clause. The nested *select* returns the maximum age of any employee in department "E21." The outer *select* returns all employees who are older than the oldest member of "E21."

ISyCL provides facilities for defining functions which are state-dependent. A state-dependent function will not necessarily return the same value each time it is applied to the same member of a domain. State functions can be used to implement generators, which generate one member of a set at a time until exhausted. Generators are important in *for_all* and *for_some* statements where it is unreasonable to construct the whole set of values.

Since state functions are nonreentrant (state-dependent), it is critical that mistakes are not made by calling one in the wrong state. Thus, ISyCL only allows state functions to be called within one of the "for" constructs, and an instance of the code is created upon entering the form and destroyed upon exit. It is also important for the author of a state function to keep it as short as possible and to avoid recursion.

2.3.4 Constraints (Functions with Triggers)

In the analyst layer, constraint refers to conditions which must be satisfied by the information system. These constraints are not functions, just descriptions.

In the systems layer, constraints are combinations of exception handlers and triggers which "enforce" constraints described in the analyst layer. By "enforce," we mean that an implementation of an ISyCL processor would actually call the triggers appropriately and then raise the exceptions. Exception handlers are functions which are called to take appropriate actions when an exception occurs.

Let's look at an analyst layer constraint on the model in Figure 2.2. The constraint is that an employee cannot be his or her own manager.

```
for_all e of entity_class:Employee
  (E#(e) <> Manager(e))
```

The constraint states that the value of the employee number (*E#*) of an employee should never be the same as the value of the *Manager* attribute. This is true since *Manager* is an inherited attribute which originates at *E#*. Thus, if the same employee number is found by both paths, then the employee is managing himself.

In the systems layer, it must also be decided when to check the constraint (trigger) and what to do if it fails (exception handler). In this case, the constraint should be checked upon addition of a new employee or upon assignment of an employee to a

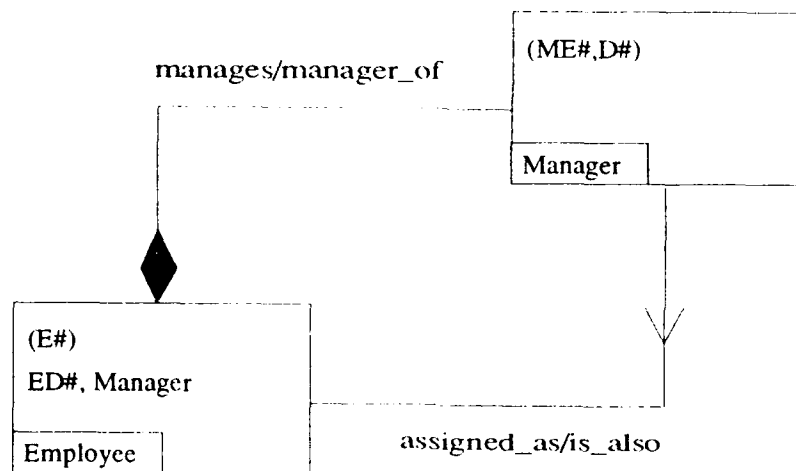


Figure 2.2 Employee - Manager Relationship

manager (modifying the *Manager* attribute of an *Employee*). If the constraint fails, a warning should be sent to the user. The exception handler would look like:

```

exception employee_managing_self (Employee e) continue
    "An employee cannot manage him- or herself."
    [print(stdout, "Employee named \s is managing him or herself." name(e));]
  
```

The "continue" at the end of the first line signifies that execution continues after the point where the exception was raised. In other words, this is not serious enough to halt execution of the function. The triggers for this example are:

```

after init of Employee e ()
    [if (E#(e) = Manager(e))
        raise(employee_managing_self(e));]

before mod of attribute_class:Manager (Employee e, employee_number new_manager)
    [if (E#(e) = new_manager)
        raise(employee_managing_self(e));]
  
```

The first trigger checks new employees and the second checks when an employee is assigned to a different manager.

The IDEF1 Slice

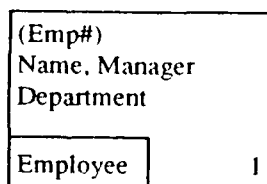
Each method has a set of meta layer ISyCL definitions used to describe models in that method. These meta layer definitions play an important part in the use of ISyCL, and careful attention should be paid when creating these definitions. It is likely that a standards committee would be formed to ensure that each method has a proper set of meta layer definitions which everyone would use.

The following is a discussion of the application of the IDEF1 metaclasses. At this point, we are not interested in how to define the metaclasses, only how to use them. Please refer to the "Analysis of Methods" report for a concise description of IDEF1 and the metamodel from which the following definitions were derived Mayer et al. [89].

3.1 Entity Classes

Entity classes are more or less buckets of attribute classes. The attribute class names describe the information kept about entities of the entity class. Entity classes also have key classes which are groups of attribute classes whose collective values uniquely identify a member entity of the entity class. Entity classes are the only "data objects" in IDEF1. All other structures like attribute classes and link classes are functions.

The *Employee* entry class might appear as follows.



An *Employee's* *Emp#* uniquely identifies that information set. *Name*, *Manager*, and *Department* are descriptive attribute classes. The employee's social security number, if kept, could also be used to uniquely identify the employee, thus adding a second key class.

The ISyCL representation of the *Employee* entity class might look like:

```
entity_class Employee
  [Emp# :      employee_number, unique;
   Name :      string;
   Manager :    employee_number;
   Department:  department_number;]
```

Note that IDEF1 does not allow “entity-valued” or “set-valued” attribute classes. Thus, the *Manager* attribute class’s value is an *employee_number*, not a pointer to the *Employee* entity. *Employee_number* and *department_number* are abstract data types which are probably subranges of *integer*.

The general syntax is:

```
entity_class <entity-class-name>
  [<owned-attrb-name> :      <attrib-value-type> [, unique];
   ...]
```

3.2 Owned Attribute Classes

There is one major problem with the *Employee* entity class above. The *Manager* and *Department* attribute classes would probably not be owned by *Employee*. These attribute classes would be inherited attribute classes.

Owned attribute classes originate at that entity class, whereas inherited attributes map back to an owned attribute class. It is important to note that attribute values are not inherited. Owned attribute classes are accessors which access a stored value, whereas inherited attribute classes map back to owned attribute classes through link classes.

Figure 3.1 is a very important figure. At the top is the IDEF1 diagram of the Employee - Manager relation. Below that is the functional mapping associated with the diagram. On the left side of the mapping are the entity classes named *Employee* and *Manager*, along with the mappings associated with the link classes *manager_of* and *is_also*. The attribute value class *employee_number* on the right side names the codomain of the attribute class *Emp#*.

Many employees map to a single manager (one-to-many link), each manager maps to an employee (one-to-one link class), and each employee has a unique *Emp#*.

The *Employee* entity class should have been defined:

```
entity_class Employee
  [Emp# :      employee_number, unique;
   Name :      string;]
```

The *Emp#* and *Name* attribute classes map specific entities to attribute values. Note that IDEF1 cannot refer to individual entities, whereas ISyCL can. To find the employee number of the *Employee e*, one would use:

$Emp\#(e)$

Before discussing more about inherited attribute classes, we need to discuss link classes.

3.3 Link Classes

Link classes describe the relationships between entity classes. Link classes are generally read from the independent to dependent entity classes. For example, "A manager manages many employees." In such a relationship, *Manager* would be the independent entity class and *Employee* would be the dependent entity class.

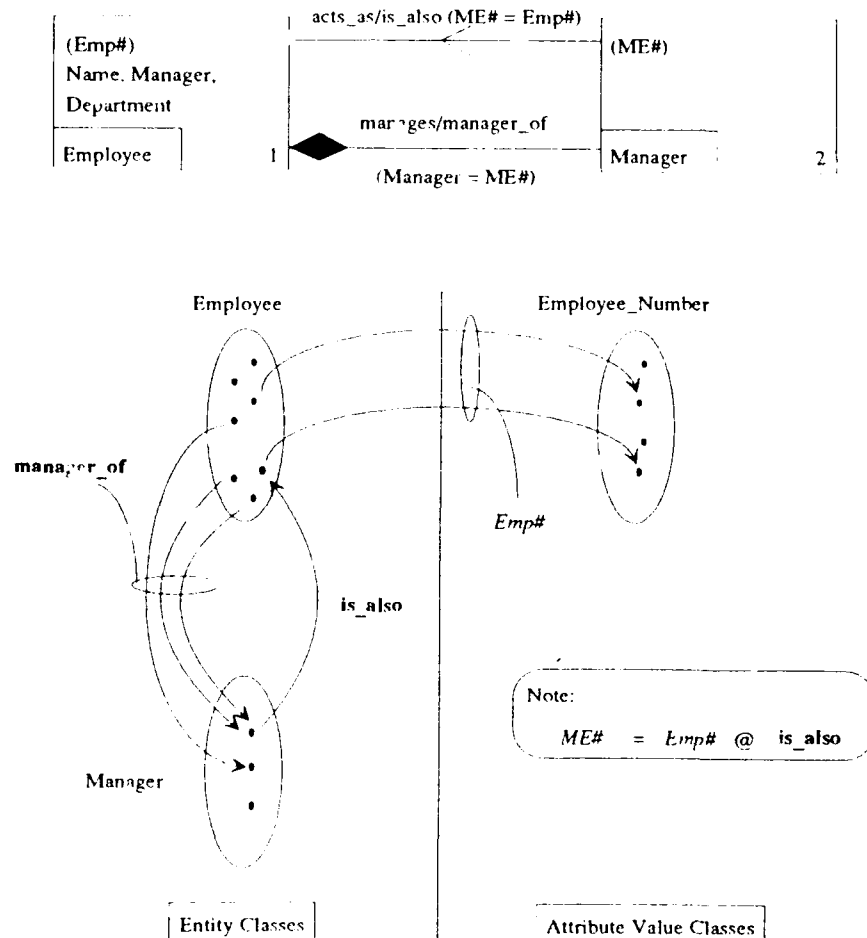


Figure 3.1 IDEF1 Attribute Class Mapping

Link classes have labels which describe the relationship. Conventional IDEF1 has one label which reads from independent to dependent. That label is the name of a function which takes two arguments, the independent entity and the dependent entity class, and returns the set of dependent entities. Such as:

$$\text{manages}(m, \text{Employee}) \Rightarrow \{e, f, g\}$$

which returns the set of employees which are managed by m . Note that m is an entity of the entity class *Manager*, whereas *Employee* is an entity class.

To enhance its use with ISyCL, a second label is added to the link class which describes the relationship from dependent to independent in a functional manner. For instance, *manager_of* would return the manager an employee works for, like:

$$\text{manager_of}(e) \Rightarrow \# \langle \text{Manager "Bob"} \rangle \text{ (the "meatball" representing the manager entity)}$$

Note that independent to dependent names do not need to be unique within a model, but dependent to independent need to be unique for an entity class. In other words, an entity class cannot participate as the dependent entity class in multiple link classes which have the same dependent to independent naming.

The cardinality of the link class determines the possible number of members of the set returned from the independent to dependent application of the link class. There are three different cardinalities: *one_to_one_link_class*, *weak_many_link_class*, and *strong_many_link_class*.

Link classes are defined like:

```
define strong_many_link_class (manager_of, Employee, Manager, (inverse manages))
```

which defines the link classes we have been discussing. *Manager_of* maps an employee to his or her manager, and *manages* maps a manager to his or her employees. The general syntax is:

```
define <link-class-type>
  ( <link-class-name>, <dep-entity-class>, <indep-entity-class>
    [, (inverse <indep-to-dep-name>) ] )
```

3.4 Inherited Attribute Classes

As was discussed earlier, inherited attribute classes map back through owned attribute classes to attribute values. The mapping is done through link classes. Inherited attribute classes are "inherited" through link classes, which means a mapping is formed from entities of the dependent entity class to entities of the independent entity class.

Pay particular attention to the note in Figure 3.1. If the entity class *Manager* has an attribute class *ME#* which it inherited from the entity class *Employee* (*Emp#* changed to *ME#* across the link), then *ME#* applied to a manager is the same as applying the link it was inherited across (*is_also*) to the manager and then applying *Emp#* to the result of the first application (which is an instance of *Employee*).

Inherited attribute classes are inherited from the key classes of independent entity classes to dependent entity classes. Depending on the cardinality of the link class, the inherited attributes may or may not participate in a key class of the dependent entity class. In one-to-one link classes, entire key classes are inherited across the link class, since the key class of the independent entity class must be capable of uniquely identifying members of the dependent entity class.

Inherited attribute classes are defined like:

```
define inherited_attribute_class (Manager, Employee, E#, is_also@manager_of)
```

which defines an inherited attribute class, *Manager*, of the entity class, *Employee*, which originates at the owned attribute class, *E#*, and is inherited through the link classes *is_also* and *manager_of*.

The general syntax is:

```
define inherited_attribute_class
  (<inherited-attrb-name>, <of-entity-class>, <owned-attrb>, <link-class-path>)
```

3.5 Key Classes

Key classes are ordered sets of attribute classes whose values uniquely identify entities of an entity class. Key classes can contain owned and inherited attribute classes. Key classes are actually just the list of attribute classes, but the usage of key classes is to find unique entities. Thus, key class accessors are defined which return unique entities of an entity class. Key class accessors are defined like:

```
define key_class_accessor (get_employee, (E#), Employee)
```

which defines *get_employee* which takes an employee number and returns an employee. The general syntax is:

```
define key_class_accessor (<accessor-name>, <key-class>, <entity-class>)
```

3.6 Usage

In most cases, the definitions discussed above will be generated from an automated IDEF1 tool. The reason to understand these definitions is for use in constraints. For

example, when a link class is added to a model, the functions which map between the entity classes will be available for use in constraints.

Given the definitions above we could write the constraint:

```
for_all e of entity_class:Employee  
  (Emp#(e) <> Manager(e))
```

The fact that *Manager* is an inherited attribute class and *Emp#* is owned is not important other than to realize that the constraint is meaningful because of the mapping. Each automated tool should clearly define the assumptions made when generating the ISyCL definitions.

IDEF1 Model with ISyCL Constraints

Now that the characteristics and language constructs of the ISyCL have been presented, an example of an ISyCL application is in order. In our example we will stay with the theme of employees. The IDEF1 model in Figure 4.1 shows the relationships between tasks, employees, and backup employees.

4.1 Model Element Representation

Let's look at the ISyCL definitions of the model elements in Figure 4.1 involving the *Employee* entity class. If this description proceeds too quickly, refer back to Chapter 3 for a description of the IDEF1 "slice." Also, remember that use with IDEF1 is only an example of the use of ISyCL, and that ER, IDEF3, or other methods could as easily have been chosen.

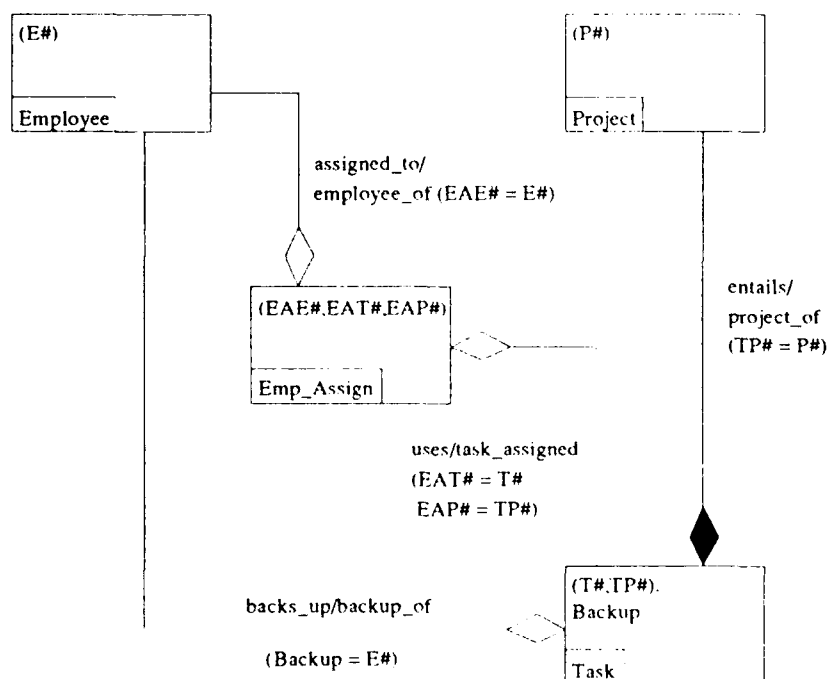


Figure 4.1 IDEF1 Example Model

First, the definitions of *Employee* and its dependents:

```
entity_class Employee
  [E# :      employee_number, unique;]

entity_class Emp_Assign;

entity_class Task
  [T# :      task_number, unique;]
```

Employee has one owned attribute class, *E#*, which must have unique values for all employees. Entity class *Emp_Assign* has no owned attribute classes, and *Task* has *T#* which serves the same purpose as *E#*.

Next, we define the link classes. *Employee* participates as the independent entity class in two link classes, *employee_of* and *backup_of*. Both are weak one-to-many link classes, meaning that it is possible that zero dependent entities participate in the relationship.

```
define weak_many_link_class (employee_of, Emp_Assign, Employee);

define weak_many_link_class (backup_of, Task, Employee);
```

The definition of *employee_of* states that *employee_of* is a weak one-to-many link class whose dependent entity class is *Emp_Assign* and whose independent entity class is *Employee*. *Backup_of* is similar.

Inherited attribute classes are the most complex definitions. *Employee* has no inherited attribute classes, but let's look at the inheritance of *E#* across the link classes just discussed.

```
define inherited_attribute_class (EAE#, Emp_Assign, E#, employee_of);

define inherited_attribute_class (Backup, Task, E#, backup_of);
```

In the first case, *E#* is inherited by *Emp_Assign* as *EAE#*. The parameters are, from left to right:

- (1) the name of the inherited attribute class (*EAE#*),
- (2) the entity class in which the inherited attribute class appears (*Emp_Assign*),
- (3) the owned attribute class from which the inherited attribute class originates (*E#*), and
- (4) the path through which it is inherited (*employee_of*).

In this case *EAE#* is inherited directly from *Employee*, but it might have been inherited across a composition of link classes.

Finally, we define the key classes by defining key class accessors. Key class accessors are used to access an entity from the entity class. The key class accessor for *Employee* is:

```
define key_class_accessor (get_employee, (E#), Employee);
```

This declares that *get_employee* is a key class accessor of *Employee*, and that the key class consists of the attribute class *E#*.

Usually an automated tool would generate these definitions from a diagram. Most of IDEF1's structures are functions. The functions are defined using conventional formatting for *function_class* instantiation which is discussed in Chapter 12.

The form:

```
define key_class_accessor (get_employee, (E#), Employee);
```

actually generates the function:

```
key_class_accessor get_employee (list_of(integer) val_list) : Employee
  "Find the entity identified by the key class attribute values."
  [loop for c of entity_class:Employee
    if (attribute_values_equal_values?((E#), val_list))
      return(c);
  end_loop]
```

saving the analyst and systems programmer a great deal of effort and confusion. There is no reason that a systems programmer need understand more than that the declaration:

```
define key_class_accessor (get_employee, (E#), Employee);
```

allows him or her to access tasks by:

```
get_employee(12);
```

which returns the entity of *Employee* with *E#* = 12. The meta layer definitions *can* be included in a header file, just like standard input/output capabilities are handled in C. Normally, meta layer definitions are associated with methods. The model definition specifies which method to use and consequently which meta layer definitions to load.

4.2 Constraints

It is necessary to ensure that an employee is not his or her own backup on a project. IDEF1 does not provide the ability to place constraints on entities within an entity class, but ISyCL does.

4.2.1 Area Expert and Analyst Layers

It is unreasonable to expect the area expert to understand all the different methods, much less ISyCL. Likely, the area expert would write the constraint like: "An employee cannot act as backup on his or her assigned task." Though this is certainly readable, it is not very precise. More formally, the area expert could write:

For all employees e , backup of task assigned to $e \nleftrightarrow e$.

This provides the analyst with a clearer understanding of the constraint.

Once the area expert(s) have been interviewed and enough information collected to create the model (in this case the IDEF1 model in Figure 4.1), the analyst would then add the constraint:

for_all a of entity_class:Emp_Assign
 (employee_of(a) \nleftrightarrow backup_of@task_assigned(a))

At this point the analyst would take the model and constraint(s) back to the area experts for validation. Though the constraint above is not as easy to read as the English sentence, it can be explained relatively easily. The "@" is the function composition operator. In this case, *backup_of* is composed with *task_assigned*.

For many applications, this is as far as ISyCL will be used. The modeler has a language extension which allows clear and precise expression of constraints that cannot be expressed in the graphical language of the method.

4.2.2 Systems Layer

On the other hand, many will seek to forge ahead and implement information systems based on the models which have been constructed. This is where we enter the systems layer of ISyCL. A database designer would now use the model and constraints to implement the information system. The modeling tool would be able to generate the ISyCL representation of the model (as discussed on page 31). There is yet more work to be done on the constraint that employees cannot be their own backups.

First, it must be decided which entity class(es) to constrain and what type of triggers are needed. As will be seen, these choices are very important. A trigger would need to be applied after an employee is assigned to a task whether as backup or not. Before an employee is assigned to a task, the backup of the task needs to be checked against the employee to make sure they are not the same person. Also, before a backup is assigned, the backup must be checked against the currently assigned employees. Thus, triggers would be checked *after* the instantiation of an *Emp_Assign* and *before* modification of both the *E#* attribute class of an *Emp_Assign* and the *Backup* attribute class of a *Task*.

There is no actual constraint statement. Constraints are a combination of triggers and exception handlers. The exception handler for the constraint that employees cannot be assigned to a task and backup of the same task might be written like:

```
exception emp_assign_also_backup (Emp_Assign a) continue
    "An employee cannot act as backup on his or her assigned task."
    [print(stdout, "~S is both assigned to task ~S and backup of the task."
        name(a), name(task_assigned(a));]
```

This statement declares that if this exception is raised, it must be passed the *Emp_Assign* which failed to meet the condition. Also, the *continue* means that execution continues after taking the action in the block. The action in this case is to print a warning to the user that an employee is assigned to a task and also assigned as the backup of the task.

The triggers are:

```
after init of entity_class:Emp_Assign a ()
    [when (employee_of(a) = backup_of@task_assigned(a))
        raise(emp_assign_also_backup, a);]

before mod of attrib backup of entity_class:Task t (Employee_Number backup#)
    [loop for a of entity_class:Emp_Assign where (task_assigned(a) = t)
        when (E#@employee_of(a) = backup#)
            raise(emp_assign_also_backup, a);
    end_loop]

before mod of attrib EAE# of entity_class:Emp_Assign a (Employee_Number emp_assign#)
    [when (emp_assign# = backup@task_assigned(a))
        raise(emp_assign_also_backup, a);]
```

The first trigger checks after instantiation of an employee assignment to see whether the employee assigned is also the backup. Of course, the closer one gets to implementation the more program-like the syntax becomes. These triggers are meant to be readable to a database designer, not the area expert.

As with any functions, triggers must declare the parameters they require. Triggers typically do not return values since they are not called by other functions. The first trigger takes no parameters, and none of the triggers return values.

The second trigger checks to see whether an employee who is to be assigned as backup on a task is already assigned to that task. This trigger requires the employee number of the proposed backup. (*Before* triggers always take the same parameters as the functions they constrain.) In this case, the system must loop through all employees assigned to the task to see whether any are the proposed backup.

Finally, the third trigger checks the employee number of a new employee assignment against the backup of the task. No iteration is required since there is only one backup on a task.

These triggers can now be used along with the definitions in the model to implement the information system. The model can actually be used for more than just analysis.

General Conclusions

Before moving on to the language reference, it is time to look back on our goals to determine how successful ISyCL is at satisfying them and to look forward to see what lies ahead for ISyCL.

5.1 Applicability

There probably is not much doubt that a need exists for a language which can express constraints on graphical methods and provide a neutral representation format for integration of methods.

Through the use of first-order logic, ISyCL provides the expressive power needed to express constraints on models. Though at times difficult, first-order logic is capable of expressing any conceivable constraint.

ISyCL's meta layer definition facilities allow for the definition of method slices which allow the modeler to easily express constraints on diverse methods. Since all models translate into ISyCL, an expert system can use the ISyCL representation as a neutral representation format for translating between methods.

5.2 Effectiveness

It may be a while until a conclusion can be drawn as to the effectiveness of ISyCL. This report and the instruction and usage by members of the modeling community are essential to the effective application of ISyCL. The syntax has been broken into layers to reduce the burden on new users, while still providing the powerful constructs needed at the systems layer and meta layer.

It is hoped that the community will provide constructive criticism which can be used to improve the language for everyone's benefit. ISyCL is not a proprietary language. It is meant to grow and change as the needs of the community change.

5.3 Reliability

ISyCL bases its reliability on simplicity of use. Much work must go into the meta-layer definitions so that the user need not be concerned over arduous details and devastating misrepresentations. If the meta-layer definitions are done properly, then the reliability is maintained through the simple interface with the user.

ISyCL also enhances the reliability of models due to the addition of constraints. Often constraints are glossed over because a method does not have the capability of expressing the constraint. At best, constraints have been provided in textual form which may be misinterpreted.

When ISyCL is taken out to the systems layer, reliability is improved through the direct application of constraints on the information system without the intervening interpretation of a database programmer. The information system can be designed and maintained within the same layer.

5.4 Verifiability

ISyCL's analyst layer syntax is meant to be as readable as possible so that the experts in the field being modeled will be able to verify constraints with minimal explanation from the modeler. By allowing the experts to verify the constraints, the chances of misinterpretation are further reduced.

5.5 Future Directions

This report will continue to evolve as ISyCL evolves. Concurrently, an ISyCL processor needs to be developed to demonstrate the usefulness of ISyCL for method integration and information system implementation. The development of an ISyCL processor will provide a testing ground for ISyCL's systems layer. It is hoped that the community will provide constructive comments which can be used to evolve ISyCL into a mature language.

Part II

Language Reference

Types

Types define data representations which are understood by the ISyCL processor.

ISyCL provides a rich variety of built-in types. It is hoped that having a rich set of predefined types will relieve some of the programming burden from users of ISyCL. Figure 6.1 shows ISyCL's built-in type hierarchy. At the root of the tree is the type *type*. *Type* is the supertype of all other types. Types in *italic* only act as supertypes of other types (typically called "mixins"). They are not sufficiently descriptive for variables to be declared to be of those types.

From our discussion in Section 1.4, it should be evident that types can be defined intensionally or extensionally. One could define the type, *my_integers*, to be the members of the set {1, 3, 2}. Thus, *my_integers* is defined extensionally. One could also define the type, *natural_number*, which describes all natural numbers.

The polymorphism of ISyCL functions allows functions to act differently given different types of parameters. Chapter 10 will introduce classes. Each type has a class of the same name associated with it. Thus, the number "1" is both *of_type* and *of_class* integer. Classes describe higher level types. Built-in types cannot be

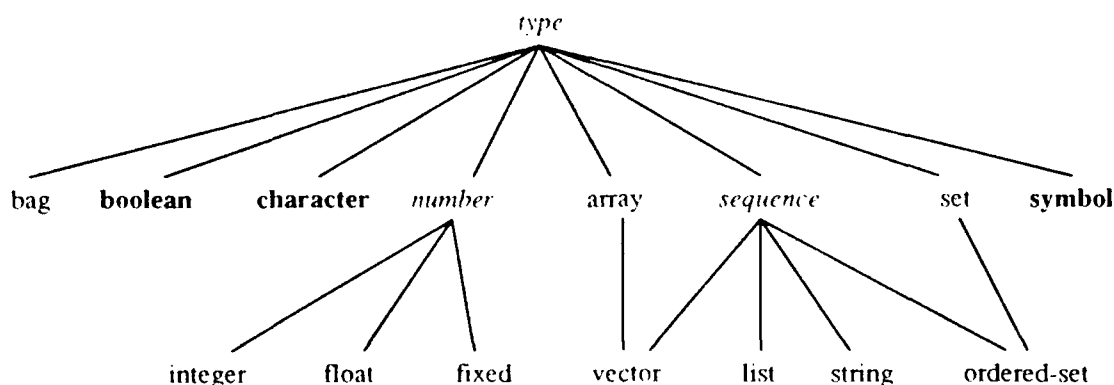


Figure 6.1 ISyCL Type Hierarchy

inherited by classes. In other words, it is not possible to define a new type of *integer*, *array*, etc. On the other hand, it is possible to define subranges of types.

Actually, only three “leaf” types are shown in the tree. *Boolean*, *character*, and *symbol* have no subtypes. The other types have system-defined functions associated with them which identify subtypes of the type. For instance, *list_of(integer)* identifies a subtype of *list* (namely, singleton lists of integers).

6.1 Type Declarations

The syntax of ISyCL requires type declarations so that the ISyCL processor can enforce strict type checking. All variables must be declared to be a defined type. Types are declared within a *with* form. Each function has an implicit *with* around its body. Thus, type declarations usually appear before the body of functions. The syntax for type declarations is:

```
<type-decl> ::= <symbol-list> : <type-form> [, init_value <expression>] ;
<type-form> ::= <type> | <subrange> | <function-call>
```

For example,

```
i : integer, init_value 3;
```

declares *i* to be of type *integer* with an initial value of 3.

In type declarations, ISyCL allows either a type, a subrange, or a function which returns a type or subrange. Given the function:

```
function int_float_list () : type
  [list_of(integer, float);]
```

the declaration,

```
x : int_float_list();
```

assigns *x* to be a list of an integer and a float. This ability allows for a flexible approach to abstract data types. As was mentioned above, many types have functions (most with the same name) associated with them which generate subtypes of the type.

6.2 ISyCL Types

6.2.1 Array

Arrays are used to store values which need to be accessed quickly no matter where they are located. All elements of an array must be of the same type, and the size of an array is fixed. Due to these restrictions, arrays can be implemented very efficiently. Arrays are also easy to use.

Arrays can be created using *make_array*, like:

```
make_array(list(5, 10), integer);
```

The first parameter must be a list of integers which are the dimensions of the array, and the second parameter is the element type. Declaration of array variables is done with:

```
a : array('5,10', integer);
```

which defines a five by ten array of integers called *a*. See Section 6.2.6.1 for an explanation of *list* and the use of the quote. To reference elements of an array, use:

```
d := a[2,8];
```

This form assigns the value of *d* to be the value of the eighth element of the second row of *a*. To assign values, use:

```
a[2,8] := 5;
```

which assigns the value of the eighth element of the second row to 5.

One-dimensional arrays are also of type *vector*. *Vectors* are *sequences*, and can be operated on similar to the other *sequence* types.

6.2.2 Bag

Bags are unordered groups of elements of the same type which may contain duplicates. There are two ways to generate bags in ISyCL. First, explicit bags can be generated using the *bag* function, like:

```
bag(1, 3, 5, 3)
```

In the second example, members are only evaluated if there is a tilde (~) in front of them. Bags can also be generated using the *select* statement. The bag of employee entities older than twenty-one would be generated with:

```
select * from entity_class:Employee
  where (age > 21);
```

Bags are similar to sets (see Section 6.2.7) other than that they can contain duplicate elements. Subtypes of bags are generated with *bag_of* (see *set_of*).

6.2.3 Boolean

ISyCL defines true to be the symbol *true* and false to be the symbol *false*.

6.2.4 Character

A character is a single character which can be described by an 8-bit ASCII character code. Since only the first seven bits are defined by ASCII, there is some implementation flexibility available for the upper 128 characters. Characters can be referenced either by preceding a single displayable character by #\ or by following the #\ with the two hexadecimal digits that correspond to the ASCII character code of that character. The character "a" could be represented as #a or #61.

6.2.5 Number

Integers, fixed-point numbers, and floating-point numbers are all based on *number*. *Number* is not an instantiable data type; it only acts as the supertype of the other number types. All numbers have ranges and units.

The purpose of ranges is not to check whether a value is within the range. Ranges are given so that the compiler can efficiently allocate memory and disk space. Many languages define numerous number types which correspond to the machine-level representations. ISyCL is not a systems programming language, consequently, there is no need to force the user to identify how to represent a number in binary.

The user can increase efficiency by providing the range of possible values. The compiler will then use the best possible representation available. Caution should be taken when assigning ranges. The range should never be too small. If a value is assigned which cannot be represented, an exception will be raised, and the program will most likely halt.

In modeling and simulation work, real-world entities are being modeled. Numbers by themselves are often not very meaningful. A velocity of 2 could represent many different actual velocities, or it could be a dimensionless velocity. Algorithms are generally based on dimensionless quantities, but values in a database are likely dimensional.

To reduce errors made by misinterpretation of units, ISyCL allows numbers to have units. Thus, values in the database will have units associated with them, and an exception can be raised if values with mismatched units are applied to one another. ISyCL will also convert between units in different systems automatically.

To keep straight which attributes or variables have units, the convention is to use capital letters in the name. For example, a velocity could be declared as:

```
V :      float(0, 1E3, 10, cm/sec);
```

which declares that *V* ranges from zero to one thousand, keeping ten digits of precision, with units of centimeters per second. It might be more convenient to declare the subtype:

```
subtype CGS_velocity := float(0, 3E10, 10, cm/sec);
```

and then declare *V* to be of type *CGS_velocity*. By declaring types like *CGS_velocity* in a standard include file, a group of engineers can count on common range, precision, and units for its data.

6.2.5.1 Fixed

Some machines have specialized hardware for fixed-point number manipulation. ISyCL provides *fixed* as a type which can be implemented by the ISyCL processor as fixed-point if the hardware supports it or as floating-point if it does not. Fixed-point variables are declared using the function:

```
fixed( <lower-bound>, <upper-bound>, <delta> [, <units>] )
```

like,

```
f :      fixed(1, 10, .001);
```

which declares *f* to be of type *fixed* with a range of 1 to 10 and a delta of 0.001.

6.2.5.2 Float

Floating-point variables are declared using:

```
float( <lower-bound>, <upper-bound>, <digits> [, <units>] )
```

like,

```
DISTANCE :      float(-1E6, 1E6, 10, cm);
```

which defines a range from -1,000,000 to +1,000,000 with ten digits of precision and units of centimeters. Note the upper case naming which signifies (but does not dictate) that *DISTANCE* has dimensions.

6.2.5.3 Integer

Integer variables are declared using:

```
integer( <lower-bound>, <upper-bound> [, <units>] )
```

like,

```
i:      integer(0, 1023);
```

which declares *i* to be an integer from 0 to 1023.

6.2.6 Sequence

Each of the following types and one-dimensional arrays (*vectors*) are *sequences*. As sequences, there are many functions which may be applied to members of these types.

6.2.6.1 List

ISyCL provides linked lists as a standard type. Though *list* is the underlying type of all list structures, the types of the elements of the list must be declared using the *list_of* function described in this section. The empty list is represented by “().”

There are two ways of defining explicit lists. The function *list* returns a list of the values of its parameters. If parameters are not to be evaluated, then a quote needs to be placed before them. A quote (') before a list returns a list without evaluating elements unless there is a tilde in front of them. For instance,

```
'(a, b, ~c)
```

returns a list of the symbols *a* and *b*, and the value of *c*; whereas,

```
list(a, b, 'c)
```

would return a list of the values of *a* and *b*, and the symbol *c*.

Lists can also be generated using *select*. For instance,

```
select * from entity_class:Employee
  where (age > 21)
  ordered asc;
```

would return the list of employees older than twenty-one in ascending order.

The implementation of *list* must keep track of the type of each member of a list. When a list is passed as an argument to a function, the type of list is determined, and the basic function which operates on that domain is applied.

There are many ways to define the types of the list elements. It is important to keep in mind that *list_of* and the other type generators return a type. Using combinations of these type generators, there are an infinite number of different types of lists.

If only a certain number of elements of the same type are to be supplied, *list_of* can be used as follows:

```
list_of(3, float)
```

which represents a list like:

```
(2.0, 5.5, 128.2)
```

If the list has a fixed number of elements of different types, use:

```
list_of(float, integer, list_of(integer))
```

which represents lists like:

```
(2.5, 2, (1, 2, 4))
```

If the list contains many elements of repeating types, the types of the elements can be defined using:

```
list_of(many, '(integer, Employee))
```

which represents a list like:

```
(1, e, 2, f, 10, g)
```

where, *e*, *f*, and *g* are employee variables. Combinations of the list constructor can be used to form very complex lists. For instance,

```
list_of@list_of(list_of(many, integer), list_of(float))
```

represents lists like:

```
((2, 3, 4), (5.1)), ((1, 2), (10.3)))
```

As will be discussed in later chapters, there are many standard functions for manipulating lists.

6.2.6.2 Ordered Set

Ordered sets are similar to lists, but ordered sets cannot contain duplicates. Ordered sets are declared using *ordered_set_of* which takes the same parameters as *list_of*. See Section 6.2.6.1 for more information about *list_of*.

Explicit ordered sets can be generated using *ordered_set* (like *list*). For example,

```
ordered_set(a, b, 'c)
```

would return an ordered set of the values of *a* and *b*, and the symbol *c*. Ordered sets can also be generated using *select*. For instance,

```
select * from entity_class:Employee
  where (age > 21)
  ordered asc
  no_duplicates;
```

would return the ordered set of employees older than twenty-one in ascending order.

6.2.6.3 String

Unlike many other languages, ISyCL provides strings as a separate data type. ISyCL does not specify the implementation of the data type. Strings are *typically* implemented as one-dimensional arrays of characters.

Strings are declared via:

```
a : string(10), init_value "Neptune";
```

or,

```
a : string, init_value "Neptune";
```

both of which initialize *a* to "Neptune." The first reserves room for ten characters, while the second reserves at least enough room for "Neptune." Similar to hash tables in Common LISP, strings in ISyCL must be able to resize automatically. Thus, the reservation of space only describes the initial number of characters which can be held without resizing.

The user should only face significant performance penalties when too little room is reserved. For instance, in the second case above, the user seems not to be concerned with future modifications. The default quantity of space reserved and the amount added when resizing are implementation dependent.

The number of characters reserved and the actual length of a string can be accessed by:

```
size(a) ⇒ 10      (# of characters reserved for a)
```

```
length(a) ⇒ 7     (# of actual characters)
```

and reference to a substring is accomplished by:

`subseq(a, 2, 4) ⇒ "ept"`

where "subseq" is short for subsequence. The functionality of strings in ISyCL will render implementations somewhat less efficient than implementations of strings in other languages, but the application of strings is much simpler in ISyCL.

6.2.7 Set

Sets are used extensively in constraints. Sets cannot contain duplicate elements, and order is not maintained.

Similar to lists, sets must have elements of declared type. The *set_of* type generator is different from *list_of* in that

`set_of(integer)`

would define a set of many integers. Since sets do not maintain order, every element of a set must be of the same type. Consequently, *set_of* is not polymorphic like *list_of*.

ISyCL provides a set-builder notation for defining explicit sets. For instance,

`{e of entity_class:Employee | age(e) > 25}`

would return the set of employees whose ages are greater than 25. Explicit sets may also be defined from:

`{1, 3, 2}`

where members are only evaluated if a tilde is placed before them, from:

`set(1, 3, 2)`

and from *select*.

6.2.8 Symbol

Symbols in ISyCL must begin with a letter and may contain characters from the set {a..z, A..Z, 0..9, #, \$, %, ?, _}. ISyCL processors convert all symbols to upper case unless a "%" is placed in front of the symbol.

6.3 Subtype and Subrange Definitions

6.3.1 Subtypes

To raise the level of abstraction, and to maintain consistent typing, ISyCL provides the ability to define new subtypes. As was described earlier, it may be beneficial to define subtypes like “CGS_velocity.” Subtypes are defined like:

```
subtype CGS_velocity := float(0, 3E10, 10, cm/sec);
```

Consequently, all velocities would have the same precision (10 digits) and units (cm/sec).

If a subset of a type is needed, then a subrange should be defined. Subtypes do not perform bounds checking (see Section 6.2.5), whereas subranges do. In Chapter 10, definition of special classes based on *entity* and *object* will be described.

6.3.2 Subranges

It is often convenient to restrict a domain to a subrange of a type. Given the definition:

```
subrange natural_number (integer i)
  (i > 0);
```

natural_number can be specified as the domain of a function. For example, say we need a function which finds employees older than a given age. We do not want the user to supply an age which is negative or zero, so we define:

```
function find_emps_older_than (natural_number n) : list_of(many, '(Employee))
  [loop for e of entity_class:Employee
    if (e > n) collect e;
  end_loop]
```

Note that this checking is done at run time. Each function has a dispatcher associated with it which determines the proper basic function to apply to the given argument. The more elaborate each domain, the longer it will take to dispatch the function. Consequently, the level of performance which is needed must be considered before using subranges.

Expressions

This chapter describes ISyCL expressions and operators. Many of the operators used in ISyCL expressions are found in modern general-purpose languages. It is hoped that there will not be too many surprises in this chapter.

Three standard expression notations could have been chosen for ISyCL: infix, prefix, and postfix. Infix places the operator between the operands, prefix places the operator before the operands, and postfix places the operator after the operands.

Infix's strength is its readability, whereas its weakness is its clumsy extension beyond binary operations. Take, for example, the addition operator "+." The addition of two numbers, say $1 + 2$, is very readable, but while $1 + 2 + 3 + 4 + 5$ is still readable, it is rather clumsy.

Infix also requires the use of parentheses for grouping. Operators are assigned various levels of precedence, like multiplication has higher precedence than addition. Thus, given $1 + 2 * 3$, the multiplication will occur first followed by the addition (giving 7). To force the addition first, parentheses would be added: $(1 + 2) * 3 \Rightarrow 9$.

Prefix is just the opposite. Adding two numbers by $+ 1 2$, is not natural, but addition is extendable to many numbers very cleanly (e.g., $+ 1 2 3 4 5$). Prefix requires a way of separating operands. For example, $* + 1 2 3 4$ is ambiguous since there are multiple ways to distribute the operands across the operators. Precedence is not an issue.

Postfix sacrifices even more readability (e.g., $1 2 3 4 5 +$), but retains extensibility and is simple to compile due to stacks. Postfix does not require any parentheses. For example, $1 2 + 3 4 * \Rightarrow 36$ is unambiguous, and again precedence is not an issue. Anyone who has used a reverse Polish notation (RPN) calculator should recognize the convenience of postfix notation.

One of ISyCL's primary goals is readability; therefore infix notation has been chosen for ISyCL expressions. Consequently, precedence rules must be given, and the user needs to use facilities like indentation to aid in maintaining readability.

7.1 Type Conversion

Often, expressions are composed of variables of different types. It would be inconvenient to force addition, per se, to operands of the same type. Addition of an integer and a float would require explicit conversion of one of the operands. ISyCL processors will perform such conversions automatically.

The rules for converting operands of expressions are adapted from those of Ada, along with additional rules for unit conversion. Detailed descriptions of type and unit conversion have not yet been published.

Note that the “=” operator will perform conversion and then check for equivalence, whereas the “==” operator will always return false if given different types of operands.

7.2 Operators

The ten different categories of operators are shown in Table 7.1.

7.2.1 Arithmetic Operators

ISyCL provides the basic arithmetic operators: “+,” “-,” “*,” and “/” along with “^” which is the exponent operator (e.g., $2^3 \Rightarrow 8$). When mixed types are used across arithmetic operators, the resulting type will be as described in Section 7.1.

Table 7.1 ISyCL Operators

Operator Type	Operators	Operator Type	Operators
Arithmetic Operators	+, -, *, /, ^	Function Composition	@
Assignment Operator	:=	Membership	in, of, of_type
Boolean Operators	and, or, not, xor, iff	Relational Operators	<, >, <=, >=, <>
Logical If	->	Vectorization Operator	!
Equivalence Operators	=, ==	Linkage Operator	<==>

Note: Spacing is not critical around operators in expressions; 2^3 will give the same result as $2 \wedge 3$.

Arithmetic operations, except for *mod* and “ \wedge ,” can also be performed on arrays of numbers. Naturally, the arrays must have the proper dimensions. The “/” operator is used to represent inverse matrices. For instance, $1/a$ would be the inverse of a , and b/a would be b times the inverse of a .

7.2.2 Assignment Operator

The assignment operator ($:=$) assigns the value of the expression on the right to the variable on the left. ISyCL supports multivalued returns. As such, one can assign values to multiple variables at once like:

```
a,b := f(x);
```

where, $f(x)$ returns two values. The first value is assigned to a , and the second value is assigned to b .

7.2.3 Boolean Operators

ISyCL supports *and*, *or*, *not*, *xor*, and *iff*. *Iff* is “if and only if” which is the same as boolean equivalence ($true = true$ and $false = false$).

7.2.4 Logical If

Logical if (“ \rightarrow ”) is an if-then construct, but it has slightly different semantics than the traditional programming language “if.” It is different in that if the condition is false, then the result of the form is true. Given,

```
if (x > 2  $\rightarrow$  y < x)
  y := x;
else
  y := 0;
```

if x is greater than two, y is checked against x . If y is less than x , y is set equal to the value of x , else y is set to zero. If x is less than or equal to two, y is set equal to the value of x . This construct is meant to be used in conjunction with quantifiers where its use is more intuitive.

7.2.5 Equivalence Operators

The “=” operator returns *true* if the operands are of equal value. The “==” operator returns *true* if the operands are exactly the same. Thus, for “==” equivalence, the operands must be of the same type, equivalent value, and in the case of entities, the same object in the database.

7.2.6 Function Composition

Functions can be composed using “@.” The resulting function has the domain of the last function in the chain and the codomain of the first function in the chain. Thus,

```
function sum_ages := sum@collect_ages;
```

defines function *sum_ages*, which takes the same arguments as *collect_ages*, and returns the same type of value as *sum*.

It is also possible to compose functions without defining a new function. For instance,

```
loop for d of entity_class:Department
  collect sum@collect_ages@employees(d);
end_loop
```

collects a list of the sums of employee ages in all departments.

7.2.7 Membership

There are three kinds of membership:

- member of a class (*of*),
- member of a sequence or set (*in*), and
- member of a type (*of_type*).

Since all built-in types have classes of the same name associated with them, members of those types can be identified with both *of_type* and *of*.

7.2.8 Relational Operators

The full set of relational operators is defined for numbers, strings, lists, and characters. Strings, lists, and sets are assigned values based on their length. If two strings are of equal length, then their characters are compared. Characters are assigned their ASCII value.

There is a special form available in ISyCL for doing range comparisons. The expression:

```
10 < x < 100
```

returns true if the value of *x* is between ten and one hundred. In many languages (like C), this form would produce meaningless results because *x* would be compared with ten and the result of that would be compared with one hundred. It just happens that in C zero represents false and one represents true, so no error occurs.

7.2.9 Vectorization Operator

Vectorization is also available. Vectorization allows every element of an array to be operated on (pseudo-)simultaneously. To use vectorization, precede the operator with a “bang” (e.g., “!*”).

7.2.10 Linkage Operator

The linkage operator allows model elements in different models to be “linked” together. In other words, through a constraint which uses the linkage operator, an expert system could track elements in different models. To show linkage between two model elements. See Section 1.1.3, Method Integration, of the introduction for more information.

7.3 Precedence and Associativity

As discussed earlier, operators must be assigned precedence and associativity to disambiguate infix expressions. Table 7.2 gives the precedence and associativity for ISyCL operators. Vectorized operations have the same precedence and associativity as their nonvectorized equivalents.

7.4 Quantifiers

ISyCL provides constructs for universal and existential quantification. These quantifiers act as predicates which determine whether all or at least one of the members of a type or set (or list) meets a condition. No order is imposed on checking the members.

The universal quantifier is *for_all* and the existential quantifier is *for_some*. The form of these constructs is like:

```
for_all e of entity_class:Employee | married?(e)
    (age(e) > 30)
```

which would determine whether all married employees are over thirty years old. If they are, it returns *true*, else *false*. The construct,

```
for_some x in {e, f, g}
    (age(x) < 30)
```

would determine whether any employee in the set is less than thirty.

Quantifiers can be nested, and pairs of set members can be tested like:

```
for_some x,y of entity_class:Employee
  (age(x) > age(y) and salary(x) < salary(y))
```

which checks all combinations of employees to see if anyone is older and makes less than anyone else. Since it is often helpful to know what entity caused a test to fail, the quantifiers allow an *on_failure* clause. The value of the *on_failure* clause is returned as the second value of the quantifier. Given,

```
for_all e of entity_class:Employee | years_with_company(e) <= 5 on_failure e
  (age(e) < 30)
```

if an employee who has been with the company no more than five years is thirty or older, then the *for_all* returns *false* and the employee entity in question.

Table 7.2 Operator Precedence and Associativity

Operators	Associativity
@	left to right
()	left to right
^, in, of, of_type	left to right
*, /	left to right
+, -	left to right
<, >, <=, >=	left to right
=, ==, <>, <==>	left to right
not	left to right
and	left to right
or, xor	left to right
iff	left to right
->	left to right
:=	right to left

7.5 Select

Select is used to collect elements. *Select* originates from SQL. To collect the bag of employees in department “E21,” one could use:

```
select * from Employee
  where dept = "E21";
```

The asterisk designates that entities are to be collected. If an attribute name were placed there, then the values of that attribute for each of the entities would be collected. There are also functions which can be used to return a particular attribute value. For instance,

```
select max(age) from Employee
  where dept = "E21";
```

would return the highest age of employees in department “E21.” On the other hand, if the oldest employees in the department were desired, then the form,

```
select * from Employee
  where dept = "E21"
  having max(age);
```

would be required. The *having* clause allows the specification of an expression which eliminates elements from the set reduced by the *where* clause. The form:

```
select * from Employee
  where (dept = "E21" and max(age));
```

would find the employees in the department which are the oldest of *all* employees.

If order is important, the *ordered by* clause can be added. Elements can be ordered by the values of an attribute of the elements. For instance, we might want to collect the ordered set of employees starting with the newest, like:

```
select * from Employee
  where dept = "E21"
  ordered by time_with_company asc;
```

The functionality of *select* can be duplicated by *loop* (see Section 8.4), but *select* is much easier to use. *Loop* cannot be used as an expression; it is a top-level iteration construct. As such, if the return value from a *loop* is desired, then a function should be written in which the *loop* is the last form executed. For instance,

```
function get_employees () : list_of(many, Employee)
  [loop for e of entity_class:Employee
    collect e;
  end_loop]
```

would return the list of *Employee* entities.

There are two primary reasons why *loop* cannot be used as an expression. First, it would be very difficult (if not impossible) for an ISyCL processor to determine at compile time the type of object returned from the *loop*. Second, using *loops* within expressions makes code difficult to follow. This will be discussed in much more detail in Section 8.4.

Statements

So far we have discussed types and expressions. Now we need to describe ISyCL statements which form the body of the language. Statements range from the relatively simple *if* statement to the elaborate *loop* facility.

8.1 Common Structures

There are many common structures used in the various statements presented below. The first structure is `<expression>`. Expressions were introduced in Chapter 7. Expressions can range from a single constant, variable, or function call up to complex expressions which perform many operations. For example,

$$(a + b) * c / f(i)$$

is a complex expression. Expressions can result in virtually any type of value. Boolean expressions are expressions which result in a boolean value, like:

$$a \leq b \text{ and } b \leq c$$

In some cases, ISyCL requires parentheses around a boolean expression. This form is called a `<condition>`. The boolean expression above would be written as,

$$(a \leq b \text{ and } b \leq c)$$

if it was to be used as a condition.

The next type of structure is the `<action>`. Actions are branches of code taken due to a condition. Actions are composed of either a single statement, or a series of statements called a `<block>`. Statements in ISyCL must end in a semicolon, and blocks of statements are enclosed in square brackets (e.g., `[]`).

8.2 Conditional Branches

The primary conditional branch is the *if* statement. *If* provides a general purpose facility for controlling actions based on conditions. ISyCL also provides the more specific *case* and *choose* statements for elegantly expressing more complex branches. *Case* and *choose* can be implemented using *if*, but the powerful expressive natures of *case* and *choose*, when appropriate, can greatly simplify complex situations.

8.2.1 The *If* Statement

The syntax for *if* is:

```
if <condition>
    <action>
else
    <action>
```

where the *else* and its action are optional. If no *else* is supplied and the condition is false, the *if* returns false. Note the difference with the *logical if* discussed earlier. *If*s can be chained together using *else*, forming the so-called *else-if* form. For example:

```
if <condition>
    <action1>
else if <condition>
    <action2>
else ...
```

It is in this manner that other complex statements can be implemented. If an action consists of multiple statements, the statements must be enclosed in brackets:

```
if <cond>
    [<statement>;
    <statement>;
    ...
    <statement>;]
```

8.2.2 The *Case* Statement

Case is used to choose the proper action based on the value of an expression. Its syntax is:

```
case <expression> [testing_with <operator>]
    (<expression> : <action>)+
    [otherwise <action>]
end_case
```

The result of the <expression> is compared with the results from each of the individual expressions using “=” or the operator specified. If the results satisfy the

condition, then the <action> is taken. After the <action>, the statement following *end_case* is executed. The superscript “+” means that there are at least one and possibly many “expression : action” pairs.

Say there were going to be ten and twenty-five year anniversary parties for employees, and we needed to collect the employee entities. This could be done like:

```
function get_anniv_employees () : list_of(many, list_of(many, Employee))
  "Find 10 and 25 year employees."
  ten_years, twenty_five_years : list_of(many, Employee);
  [loop for e of entity_class:Employee
    case time_with_company(e)
      10 : collect e into ten_years;
      25 : collect e into twenty_five_years;
    end_case
    finally return(ten_years, twenty_five_years);
  end_loop]
```

This function returns a list of two lists which contain the ten year and twenty-five year employee entities. The *case* statement in the middle of the function directs ten year employees into one list and twenty-five year employees into another list.

8.2.3 The Choose Statement

The *choose* statement is used when there are multiple conditions which may apply. The syntax is represented by:

```
choose [first <boolean-constant>
        | until <boolean-constant>
        | all <boolean-constant>]
  (<condition> : <action>)+
end_choose
```

By default, *choose* finds the first condition which is true and takes that action. It then skips the rest of the conditions and continues with the first statement following the *end_choose*. If no condition is satisfied, *choose* returns false, else *choose* returns the value of the last statement executed.

```
choose
  (weather = "snow") : print(stdout, "No Game");
  (number_of_teams = 1) : print(stdout, "Game Forfeited");
  (weather = "rain") : print(stdout, "Game Delayed");
end_choose
```

There are actually six possible cases of *choose*. The one described above refers to the “first true” case, in other words, the case where conditions are checked until one is true and that action is taken. There is also a “first false” case which would check until a condition fails and take that action.

The “*until <boolean-constant>*” case checks until a condition is the same as the one supplied, taking all actions up to that point. Finally, the “*all <boolean-constant>*” case is where all conditions are checked, and actions for conditions which return the same as the boolean supplied are taken. For example,

```
choose all true
  (age(e) >= 65) : print(stdout, "\nSenior Citizen");
  (married?(e)) : print(stdout, "\nMarried");
  (years_with_company(e) >= 25) : print(stdout, "\nAt least 25 years of service");
end_choose
```

would print messages for all of the conditions which are true.

8.3 Unconditional Branches

The nature of applications written in ISyCL should preclude the use of unconditional branches. The primary use of unconditional branches is error handling. If an error occurs, it is often difficult to gracefully exit to the error handler. Therefore, a *goto* is often used to directly jump to the error handler. ISyCL does not support *goto*.

For fatal errors, ISyCL provides an *error* function which prints a message and terminates gracefully. For nonfatal errors, ISyCL provides for declaration of exception handlers. The *with* form is wrapped around the possibly error-prone code, and *exception* handlers are declared within the *with*. Exceptions propagate up the caller stack until a handler is found, or the system handles the exception. An implicit *with* is wrapped around all functions.

The syntaxes of the *with* and *exception* forms are:

```
with <decl>*
  <action>

exception symbol <parm-list> { continue | end }
  <action>
```

where <decl> is either an exception or trigger definition or a type declaration. For instance,

```
with emp : Employee;
  exception birthday (Employee e) continue
    print(stdout, "Happy Birthday \s!" name(e));
  [...]
```

creates a local variable *emp* which refers to an *Employee* and defines an exception handler which prints a happy birthday message. The only place this variable and exception handler can be accessed from is within the action of the *with*.

The user can raise an exception by:

```
raise ( exception-name [, <arguments>] );
```

which transfers control to the action associated with that exception. Given the example above, one could raise:

```
raise(birthday, e); -- e is an employee
```

within the action of the *with*.

There are two options for continuation of the execution after the action; execution can proceed from where the exception was raised (*continue*) or execution can resume after the *with* block (*end*).

Exceptions can be used for more than just error handling. For instance:

```
function common_elements? (integer a[],b[]) : boolean
  exception common_element () end
  [print(stdout, "\i is a common element." a[i]);
   true;]
  [loop for i from 1 to first(dimensions(a))
   loop for j from 1 to first(dimensions(b))
     if (a[i] = b[j])
       raise(common_element);
     end_loop
   end_loop]
```

determines whether there are any common elements in integer arrays *a* and *b*. If there are, it raises the exception which prints a message and returns *true*, else the loops terminate normally and returns *false* (the value of the last comparison propagated out of the *loops*).

For efficiency, it may be desirable to make the assumption that values are going to be correct, and then handle any exceptions that might occur. *Continue* allows just that. The following is an example of the use of *continue*:

```
function average_ages (set_of(Employee) employees) : integer
  "Find average age of a set of employees. Protected against crashing due to employees not hav-
  ing been assigned ages."
  exception null_not_integer () continue
  [print(stdout, "Employee, \s, has no age. Replacing with \
    zero and continuing.", name(e));
   return(0);]
  [loop for e in employees
   sum age(e) into sum_of_ages;
   count e into number_of_emps;
   finally (sum_of_ages / number_of_emps);
  end_loop]
```

If an employee has not been assigned an age, his age will be *null*. When *null* is summed with an integer, a *null_not_integer* exception will be raised. Standard exceptions have defined protocols. *Null_not_integer* requires an integer value be returned if execution is to continue.

Note that if no exceptions occur, the function runs without a lot of error checking. Also, contrary to Ada, exceptions are not associated with functions, but blocks. Thus, it is possible to have different handlers for exceptions occurring at different places in the function.

8.4 Iteration with *Loop*

Loop is by no means simple. *Loop* is derived from the loop macro of Common LISP. *Loop* is extremely powerful, and it is designed to allow for readable code. Let's start simple. The following are just two samples of loops which sum the integers from 1 to 100:

```
loop for i from 1 to 100
  sum i;
end_loop

loop for i in {j : integer | j >= 1 and j <= 100}
  sum i;
end_loop
```

The steps taken in each iteration of the first loop can be controlled like:

```
loop for i from 1 to 100 by 2
  sum i;
end_loop
```

which would sum the odd numbers from 1 to 99. If the steps are in the negative direction, use:

```
loop for i from 99 down_to 1 by 2
  sum i;
end_loop
```

Sum is one of several accumulation identifiers. The full list is:

- collect[ing]
- append[ing]
- count[ing]
- sum[ming]
- max[imize]
- min[imize]

Thus, the oldest employee could be found by:

```
loop for e of entity_class:Employee
  max age(e);
end_loop
```

Iteration can be done in parallel by specifying multiple for-clauses. The list containing alternating positive and negative values converging from -100 and 100 to zero could be constructed by:

```
loop for p from 100 down_to 0
  for n from -100 to 0
    appending list(p, n);
end_loop
⇒ (100, -100, 99, -99, ..., 1, -1, 0, 0)
```

Better yet, multiple accumulators can be used:

```
loop for p from 100 downto 0
  for n from -100 to 0
    collect p;
    collect n;
end_loop
```

which saves us from creating those intermediate lists. Different accumulators can be mixed, but only those compatible with one another. *Collect* and *append* can be mixed, as can *count* and *sum*, and *max* and *min*.

Averages are simple with:

```
loop for e of entity_class:Employee
  counting e into counter;
  summing age(e) into summation;
  finally return(summation / counter);
end_loop
```

Finally specifies the operation to perform on exit from the loop; the value of which is not the return value of the loop unless specified. Loops can also continue until a condition as in:

```
loop for e in (select * from entity_class:Employee
               order by age asc;)
  count e into number_of_emps;
  sum years_with_company(e) into total_years;
  until (total_years > 100);
  finally return(number_of_emps);
end_loop
```

which finds the number of youngest employees it takes to accumulate better than 100 total years with the company. Note the use of the SQL *select* within the loop. The *until* clause determines when to halt the loop.

*If*s can also be used within a *loop*, like:

```

loop for e of entity_class:Employee
  if (age(e) < 30)
    count e into youngsters;
  else
    count e into veterans;
  finally return(youngsters, veterans);
end_loop

```

which returns two values, the numbers of youngsters and veterans.

Instead of using loops as expressions, a new function is written whose return values are those of the *loop*. For instance, say we wanted to check the average age of employees against each employee's age and count those who were older than average. We might have written (*incorrectly*):

```

loop for e of entity_class:Employee
  if (age(e) > (loop for e of entity_class:Employee
    counting e into counter;
    summing age(e) into summation;
    finally return(summation / counter);
  end_loop))
    count e;
end_loop

```

⇒ SYNTAX ERROR! LOOP USED AS EXPRESSION.

Leaving aside efficiency concerns (which are numerous with the above form), this form could provide the number of employees older than the average. Note how messy the form is. It is also difficult to determine what the type of return value from the inner *loop* will be. A better (and *correct*) way to accomplish the task would have been to define the function:

```

function avg_emp_age () : float
  [loop for e of entity_class:Employee
    counting e into counter;
    summing age(e) into summation;
    finally return(summation / counter);
  end_loop]

```

and then write:

```

loop for e of entity_class:Employee
  if (age(e) > avg_emp_age())
    count e;
end_loop

```

This form is much clearer.

Functions

For the most part, ISyCL functions are similar to functions in other programming languages. One important difference between ISyCL and many object-oriented languages is that ISyCL does not differentiate between functions and methods.

Object-oriented languages often refer to functions which behave differently depending on the object to which they are applied as “methods.” This primarily originates from the fact that most object-oriented languages evolved from previous languages and needed to maintain compatibility.

All ISyCL functions can exhibit polymorphism (different behavior across different domains). Also, ISyCL functions have multivalued domains and codomains. In other words, a function can accept multiple values and return multiple values.

The syntax for function definitions is:

```
{function | function_class} symbol
  {<func-assign> | <func-def>}

<func-assign>    ::= := <function-comp> ;

<func-def>       ::= <parm-list> [ : {symbol | ( <symbol-list> )?
                               [<doc-string>]
                               [ ( <attr-assign-list> ) ]
                               <decl>*
                               <block>
```

where,

- *function_class* is a symbol which names a class of functions,
- *symbol* is the function name,
- <parm-list> is the formal parameter list (see Section 9.1),
- <doc-string> is a string which describes the function,
- <attr-assign-list> is attribute value assignments for this function instance,
- <decl>* is variable, exception, or trigger declarations, and
- <block> is the body of the function.

A general function to add a list of integers could be written,

```
function sum (list_of(many, integer) num_list) : integer
  "Adds n integers."
  [loop for n in num_list
    sum n;
  end_loop]
```

which would be applied like:

```
sum('(1, 2, 3, 4)) ⇒ 10
```

Functions can return multiple values using lists. For example,

```
function collect_ages () : list_of(many, integer)
  [loop for e of entity_class:Employee
    collect age(e);
  end_loop]
```

returns the list of employee ages. ISyCL also allows multivalued returns, such as,

```
function generate_gross_and_deductions () : (list_of(many, integer), list_of(many, integer))
  g,d : list_of(many, integer);
  [loop for e of entity_class:Employee
    collect gross(e) into g;
    collect deductions(e) into d;
    finally return(g, d);
  end_loop]
```

which could be used with:

```
function collect_net_incomes (list_of(many, integer) g,d) : list_of(many, integer)
  [loop   for gross in g
        for deduction in d
          collect (gross - deduction);
  end_loop]
```

like:

```
collect_net_incomes(generate_gross_and_deductions())
```

This application would return the list of net incomes for all employees. Using the *sum* and *collect_ages* functions defined earlier, the statement,

```
sum(collect_ages())
```

would return the sum of the ages of employees.

It is also possible to compose functions for later use. For instance, if we planned to sum employee ages often, we could declare:

```
function sum_ages := sum@collect_ages;
```

Now *sum_ages()* will return the sum of the ages of the current employees. Functions can be scoped locally like variables. Thus, if employee ages are changing often within a function it would be better to define the local function *sum_ages()*, whereas if the ages are static it would be better to use a local variable instead.

9.1 Parameter Lists

In function definitions, the types of the parameters are given by a parameter list (<parm-list>). Parameter lists identify the type of the variables and the names of the variables. A function which requires an integer and a float would have the parameter list:

```
(integer i; float f)
```

Since several parameters in a row can have the same type, one can write,

```
(integer i,j,k; float f)
```

which declares that the function requires three integers and then a float. There are some special forms which allow complex types to be passed. To specify that a function accepts a list of integers, a set of floats, and a two-dimensional array of integers, use:

```
(list_of(many, integer) i_list; set_of(float) f_list; integer a[][])
```

If parameters are to be passed by reference instead of by value, then an asterisk should be placed in front of the variable name, like:

```
(list_of(many, integer) *i_list)
```

Complex data structures are often passed by reference, while simpler data structures tend to be passed by value. Pass by value means that the formal parameter receives a copy of the value of the argument, while pass by reference means that the formal parameter is a synonym of the argument. The following is a rather contrived example:

```
function print_ages ()
  i :      list_of(many, integer), init_value (select age from entity_class:Employee;)
  [append_sum(i);
   print(stdout, "The employee ages are: \i). The sum is: ~i." i - first(i), first(i))]
```

```

function append_sum (list_of(many, integer) *i_list)
  j : integer;
  [loop for i in i_list
    sum i into j;
    finally push(i, list(j));
  end_loop]

```

The list of employee ages is collected into *i*. *Append_sum* is then applied to *i*. Instead of copying the value of *i*, *i_list* is treated as a synonym of *i*. The sum of the ages is determined and then pushed on to the front of the list. Back in *print_ages*, the list of ages and the sum are printed.

There are special keywords allowed in parameter lists which control the evaluation and collection of arguments. The first, *"e*, is used when the function needs a parameter which is not to be evaluated. Given,

```

function sum_ages (&quote symbol given_class) : integer
  [loop for g of given_class
    sum age(g);
  end_loop]

```

the call,

```
sum_ages(Employee);
```

would return the sum of all employees' ages. Note that *Employee* did not have to be quoted. The keyword *"e* can be followed by *symbol*, *()*, or *{}*, meaning a symbol, a list of symbols, or a set of symbols, respectively.

The next keyword is *&rest*, which collects as many parameters into a list of the type specified. For instance, given,

```

function sum (&rest integer i_list) : integer
  [loop for i in i_list
    sum i;
  end_loop]

```

the call,

```
sum(1, 2, 3, 4);
```

would return the integer "10."

9.2 Function Attribute

ISyCL is similar to modern object-oriented languages such as C++ and the Common LISP Object System in its overloading of functions (polymorphism). The behavior of a function is dependent on the type of its arguments (domain). Based on the

domains of the arguments, the proper basic function is applied to the arguments and then returns a member of its codomain.

Function is similar in level to *object*. *Function* is a *function_class*, and all other *function_classes* inherit from *function*. The simpler type of functions are called basic functions. Basic functions cannot be defined directly, but are instead generated from the instantiation of functions.

Each basic function has the following attributes:

- domain: List of the types of its parameters.
- codomain: List of the types of its return values.
- reentrant?: Reentrant functions are not state dependent.
- doc_string: String describing the purpose of the function.

Note that basic functions do not have names; they are accessed through the routing list of functions. Functions have the following attributes:

- print_name: The name which represents the function.
- routing_list: List used to access basic functions.

When a function is applied to an argument, the type of the parameter is checked and then the basic function with the proper domain is applied to the argument.

The function definition:

```
function square (integer i) : integer
    "Multiply an integer by itself."
    [i * i]
```

reads, "the function square applied to integer i returns an integer." The documentation and body are optional. It seems that all function definitions would require a body, but functions can be composed of other functions; thus they do not have their own bodies. Given these simplifying functions,

```
function codomain (function f, list_of(many,type) t) : list_of(many, type)
    [codomain@basic_function(f, t);]

function doc_string (function f, list_of(many, type) t) : string
    [doc_string@basic_function(f, t);]
```

the attributes of *square* can be accessed by:

```
domains(#'square) => (integer)

codomain(#'square, '(integer)) => integer

doc_string(#'square, '(integer)) => "Multiply an integer by itself."
```

The “#” notation means we want the function associated with the symbol, not the value of the symbol. As we noted earlier, ISyCL allows overloading of function names. Thus, one could later define:

```
function square (float f) : float
    "Multiply a float by itself."
    [f * f]
```

with the following results:

```
domains(#'square) ⇒ (integer float)
codomain(#'square, '(float)) ⇒ float
doc_string(#'square, '(float)) ⇒ "Multiply a float by itself."
```

Now *square* has two domains, *integer* and *float*. Given a *float*, *square* returns a *float*.

Functions can also have other attributes. These attributes are declared in the *function_type* definition. Values are assigned to these attributes in the <attr-assign-list> section of the function definition.

For example, say we wanted to assign a function to be nonreentrant. We would write:

```
function example ()
    "This is just an example."
    (reentrant? := false;)
    [print(stdout, "Actually there is nothing non-reentrant about this function.");]
```

Now, this function could only be called within a loop, and an instance of the code would be created upon entrance to the loop and destroyed upon exit.

This chapter has described the foundations for ISyCL functions and the *function_class*, *function*. In Chapter 12, *function_classes* will be discussed in more detail.

9.3 Macros

Macros are special types of functions which are expanded in place at compile time. This expansion causes there to be less overhead at run time. The primary use for macros in ISyCL is in the meta-layer, but they may also be used in the systems layer.

Macros look similar to functions. Their syntax is:

```
macro symbol <symbol-list>
    [<doc-string>]
    <block>
```

The macro to add any number of integers might have been written:

```
macro sum (num_list)
  "Adds n integers"
  [loop for n in ~num_list
    sum n;
  end_loop]
```

Compare this form with the function on page 65. The code in the block is not evaluated unless it is preceded by a tilde. Thus, in this case, *num_list* would be replaced by a form which will produce the list of integers to be summed, but other than that, the block would be inserted directly into the code. Consequently,

```
function sum_ages (list_of(many, age) age_list) : integer
  [sum(age-list);]
```

would expand to:

```
function sum_ages (list_of(many, age) age_list) : integer
  [loop for n in age_list
    sum n;
  end_loop]
```

Note that the tilde only applies to the symbol immediately following the tilde. If an expression or function call needs to be evaluated, parentheses must be placed around the form to be evaluated. For example,

```
~func(arg)
```

is different than:

```
~(func(arg))
```

The first evaluates *func* (to determine what function to eventually apply to *arg*), whereas the second applies *func* to *arg*. Many examples of macro usage are shown in Chapter 12.

What is wrong with the *sum* macro? Remember that *loop* cannot be used as an expression. Since macros just expand in place at compile time, if *sum* were used as an expression, a compile-time error would result. For example,

```
function over_thousand? () : integer
  "Is sum of ages greater than 1000?"
  [if (sum('(select age from entity_class:Employee order)) > 1000)
    print(stdout, "Sum of ages is greater than 1000.");]
```

would expand to:

```
function over_thousand? () : integer
  "Is sum of ages greater than 1000?"
  [if (loop for n in (select age from entity_class:Employee order)
      sum n;
      end_loop > 1000)
      print(stdout, "Sum of ages is greater than 1000.");] ⇒ ERROR!
```

Loop is used as an expression; consequently, an error is reported.

Classes

ISyCL provides two special classes called *entity* and *object*. Entities represent the information kept in an information system about real or abstract objects like employees, managers, etc. Objects represent temporary objects which are of use in object-oriented programming (e.g., a window on the screen). Both classes have attributes which describe the traits of entities or objects of that class. For instance, a car could have the attributes make, model, year, etc. Attribute values of entities are kept in a database while attribute values of objects are kept in memory. By taking advantage of the polymorphism allowed by functions in ISyCL, objects and entities can take on behavioral aspects.

10.1 Objects

Object classes can be assigned user-defined attributes. Built-in classes, such as *integer*, have a fixed set of attributes (e.g., *range*). Instances of built-in classes are *objects*, though most are system-defined. In other words, there is no need to create the integer “1”; it has already been defined by the system.

The definition of *object_class*, *circle*, might look like:

```
class circle (graphic-object)
  [(center:    location;
    radius:    integer(0, 1023);)
   fill-color: color, init_value 'blue;]
```

along with:

```
subtype x_dimension := integer(0, 1023);

subtype y_dimension := integer(0, 767);

subtype location := cross_product(x_dimension, y_dimension);

subtype color := symbol in { 'red, 'blue, 'green, 'yellow, 'white, 'black};
```

where *graphic_object* is a class based on *object*. The following block describes the attributes of the class. Attributes within the parentheses in the block are designated as “required.” In other words, they must be given values upon instantiation.

Besides specifying the type of each attribute, there are many other options for attributes. Attributes can be designated as *read_only* and *non_null*, and can be given an *init_value*. *Non_null* means that the attribute must have a value. *Init_values* are default values assigned to attributes upon initialization.

Objects are instantiated using “make” functions:

```
make-circle('(100, 100), 200, fill_color, 'red);
```

“Make” functions like *make_circle* are generated upon definition of the object class. Since objects do not have keys, there are no “get” accessors for objects. The “make” functions return a pointer to the object. If that pointer is lost, then the object is lost. Types, like *integer*, do not have “make” functions.

10.2 Entities

Entity classes are different from other classes in that they define “database objects” called entities. These entities are persistent across sessions, and can be used to facilitate communications between systems.

The definition of a *Car* class could be:

```
class Car (entity)
  [(registration_number: integer, read_only;
   make: string, inverse cars_of_make, read_only;
   model: string, read_only;
   year: integer, read_only;
   locked?: boolean, non_null, init_value true;]
  ('(registration_number))
```

This states that *Car* is a class based on *entity*. In this case, the required attributes can only be given values upon instantiation, since they are all *read_only*.

Besides the standard attribute options, entity attributes can have inverse accessors which create the set of entities which have a given value of an attribute. In this example, *cars_of_make* returns the set of cars with a specific make.

The first attribute of *Car*, *registration_number*, is the sole attribute of the first key of *Car*. *Car* does not have any other keys. Keys are used to access unique entities. Entity classes which do not have any keys are not instantiable, but they can be used as “mix-ins” for instantiable entity classes. Key attributes must be given values at instantiation time.

A subclass of *Car*, *Sports_Car*, could then be defined:

```
class Sports_Car (Car)
  [top_speed:      integer;]
```

To make it clear to the reader what objects are stored in memory versus in the database, instantiations of entities are different from objects. The instantiations of *Car* and *Sports_Car* would look like:

```
insert into Car
  (registration_number, make, model, year, locked?)
values '(1200054, "Chevy", "Nova", 1986, false);

insert into Sports_Car
  (registration_number, make, model, year, top_speed)
values '(230005, "Nissan", "280ZX", 1989, 160);
```

Note the quote in front of the list of values to be assigned to the attributes. ISyCL allows any sort of list accessor to follow *values*. In other words, it could be a function call which returns a list, a symbol bound to a list, or an explicit list as shown. Multiple lists can be provided (separated by commas). Each list of values generates an entity.

Since entities are persistent across sessions, there must be a way to access entities in the database. "Get" accessors are used to retrieve handles for entities in the database. Handles are analogous to pointers. A handle points to an entity in the database. For example,

```
my_car :    Car, init_value get_car(1200054);
```

declares that *my_car* is of class *Car* and has as its initial value the *Car* with registration number "1200054." The attribute values of the instance of *Car* pointed at by *my_car* are in the database. In "Get" accessors, values for the key attributes must be specified in order of definition. Handles have states of open and closed. If another process has deleted an object pointed to by a handle, an error will occur when the handle is used.

One can also use cursors to access entities in the database. For instance:

```
declare locked_cars cursor
  for select * from Car
  where locked?;
```

Cursors are different from handles in that they return a set of entities or attribute values, or a distinct entity or attribute value. Given the above cursor declaration, a set of cars is referenced by *locked_cars*.

A way to access the make of the car whose registration number is “1200054” using a cursor would be:

```
declare my_car cursor
  for select distinct * from Car
    where registration_number = 1200054;
```

and then to use *make(my_car)*.

10.3 Behavior

Polymorphism can be used to add behavior to entities and objects. It has been found that people find it easier to program in terms of objects. For instance, cars can be locked. It is debatable whether the person locks the car or whether the locking mechanism in the door locks the car upon request from the person. Given the latter interpretation, that the car locks itself, we could write a function to lock the car:

```
function lock (Car c) : boolean
  "Lock the car, and return the previous value."
  [if (not locked?(c))
    [locked?(c) := true;
     false;]
   else
     true;]
```

Constraints

Given the name, Information Systems Constraint Language, it should not be surprising that constraints are an important part of ISyCL. Constraints are a combination of special functions called triggers and exception handlers. Triggers are functions which are called upon the occurrence of an event. Exception handlers were discussed in Section 8.3. A constraint is the combination of a trigger (or triggers) and an exception handler which is raised if a condition is violated.

One example of a constraint is the constraint that the value of an employee entity's salary attribute cannot be negative. Anytime an employee's salary is modified, the value is checked. If the value is negative, then an exception is raised which keeps the transaction from occurring. This constraint would be written:

```
exception negative_salary (Employee e) end
  print(stdout, "Employee \a cannot be assigned a negative salary.", name(e));

before mod of attrib salary of Employee e (fixed(0, 1E6, .01) new_salary)
  [if (new_salary < 0)
    raise(negative_salary, e);]
```

The first form defines the exception handler and the second defines the trigger. If an employee is assigned a negative salary, the exception is raised, a message is displayed, and the assignment does not take place. If, instead of *end*, we had written *continue* as the type of exception, then the assignment would be made.

11.1 Triggers

Triggers are very flexible. Triggers can be placed on any function, and they can be triggered before or after the function. It is also possible to define triggers which enable triggers during the execution of a function. Before going deeper into *during* triggers, let's look at a real-world example.

To ensure the safety of lathe operators, they must keep both hands on the lathe control panel during operation. If the operator's hands leave the buttons, the lathe stops.

In ISyCL terms, there is a *during* constraint which enables triggers during the operation of the lathe. The triggers raise an exception if the buttons are released. The exception handler stops the lathe. Any time other than during operation of the lathe, the state of the buttons is not watched.

The ISyCL description of the lathe constraint would look like:

```
during operate of Lathe l ()
  exception lathe_buttons_released (Lathe l) continue
    halt(l);
  before mod of attribute:button_state (Lathe l; new_state)
    [if (new_state = "released")
      raise(lathe_buttons_released, l);]
  [print(stdout, "If lathe buttons are released, lathe will halt!");]
```

This form defines a *during* trigger on the function *operate* of *Lathe*. First, it defines the exception handler which, when raised, halts the lathe. Next, a *before* trigger is placed on the modification of the *button_state* attribute of the specific *Lathe* upon which *operate* was called. If *button_state* is to be set to "released," the trigger raises the exception which halts the lathe. Finally, before *operate* is called, the message "If lathe buttons are released, lathe will halt!" is displayed.

There is nothing wrong with a trigger taking an action instead of raising an exception as long as execution is to *continue*. Exceptions are used to gracefully exit a function.

11.2 Constraints

One usage of constraints is to allow information systems to express constraints upon entities in a database. ISyCL can work in tandem with other graphical languages to describe the conceptual schema of an information system. Chapter 1 described some examples of constraints written for IDEF1 models. Graphical languages can be used to their limits, and then the remaining constraints can be written using ISyCL. Automated tools for the methods can autogenerate the ISyCL representation of the information expressed in the diagrams.

Constraints can also be used in general applications. Since constraints can be applied to any function, they provide a powerful mechanism for general applications. As will be shown in Chapter 12, a constraint is used in conjunction with a *function_class* to implement attributes for classes of a metaclass in which the attributes are not actually stored, but instead derived. Thus, from an external point of view, the user need not be aware of whether an attribute is derived or stored.

An example constraint is taken from the Common LISP Object System. Classes can act as subclasses of a superclass and as the superclass of subclasses. See Figure 11.1 for an IDEF1 model of this relationship. Each time a new *Sub_Super_Pair* is created,

the following constraint is checked to determine if there are any loops in the hierarchy. The constraint and related function are:

```

exception Superclass_is_Subclass (integer superid) end
"The superclass is the same as the subclass or a subclass of the subclass."
[print(stdout, "\s is a subclass of itself." superid);]

after init of Sub_Super_Pair ss ()
[if (subid(ss) = superid(ss) or
    superclass?(subid(ss), superid(ss)))
    raise(Superclass_is_Subclass, superid(ss));]

function superclass? (integer super, integer sub) : boolean
  "Is super the cid of the superclass of the class whose cid is sub?"
  [for_all s of entity_class:Sub_Super_Pair
    (superid(s) = super and subid(s) = sub)
  or
  for_all c of entity_class:Class
    (superclass?(super, cid(c)) and superclass?(cid(c), sub))]
```

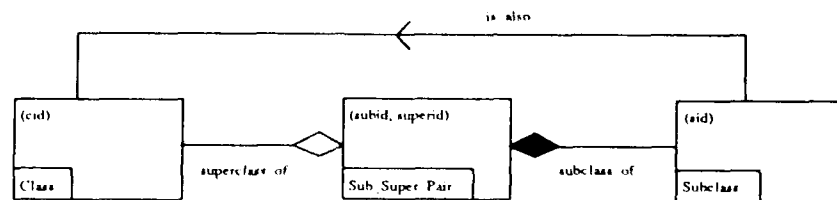


Figure 11.1 Subclass-Superclass Relationship

There are two primary conditions in this constraint. First, the *subid* and *superid* of the *Sub_Super_Pair* must be different. Second, the subclass referenced by *subid* cannot be a superclass of the class referenced by *superid*.

The predicate *superclass?* takes two class IDs and determines if the class referenced by the first is the superclass of the class referenced by the second. First, a direct relationship is checked for, then the hierarchy is checked recursively for indirect relationships.

Metaclasses

This chapter covers difficult material and is not meant for general consumption. This chapter describes constructs which can be used to customize ISyCL for particular methods. It is proposed that the experts in a method get together and define the metaclasses for “their” method. Those definitions would then be distributed to modelers who are using ISyCL with that method.

Consequently, it is unlikely that most people will need to understand the constructs in this chapter. However, this chapter will provide interested readers with a better understanding of ISyCL in general, since many of ISyCL’s constructs can be defined using constructs in this chapter.

Metaclasses define classes of classes. Metaclasses can be hierarchical, and constraints can be applied to metaclasses, the classes they describe, or the attributes of those classes. Currently, there are two classifications of classes, *function_classes* and *object_classes*. Though these two categories are sufficient now, in the future there may be a need for other classes of classes.

12.1 Object Classes

An *object_class* defines a class of data type. For example, the *entity_class* of IDEF1 is an *object_class*, which could be defined like:

```
object_class entity_class () persistent, no_inheritance, no_key
    [owned_attributes :      class_attributes (owned_attribute);
     inherited_attributes :  set_of(inherited_attribute);
     key_classes :          set_of(key_class);]
```

This definition starts by declaring that *entity_class* is a metaclass based on *object_class* (as opposed to *function_class*). Next, the series of key words declares that entities are persistent (stored in the database), there is no hierarchy amongst *entity_classes*, and *entity_classes* do not have keys (*key_classes* are defined later).

The block describes attributes of the metaclass and attributes of the class. The first line declares the *class_attributes* of this metaclass are called *owned_attributes*, and

the attributes are of type *owned_attribute*. *Class_attributes* must come first in the listing, and each group of *class_attributes* causes an additional block to be added to the definition of instances of the metaclass. For instance, the *entity_class*, *Employee*, would be defined by:

```
entity_class Employee
  [Emp# :    employee_number, unique;]
```

To retrieve the list of owned attributes of an *entity_class*, one would use:

```
owned_attributes(Employee) => (Emp#)
```

We still need to define inherited attributes, but to define inherited attributes we first need to describe *function_classes*.

The general syntax for *object_class* definitions is:

```
object_class symbol ( <symbol-list> ) [<obj-class-keywords>]
  [<doc-string>]
  <attribute-block>

<obj-class-keywords> ::=    <obj-class-keyword>
                           <obj-class-keyword>, <obj-class-keywords>

<obj-class-keyword> ::=    persistent | no_inheritance | no_key

<attribute-block>    ::=    [ <attr-list> ]

<attr-list>          ::=    <class-attr-decl>* <attr-decl>*

<class-attr-decl>    ::=    symbol : class_attributes ( symbol )
```

Object_classes can also describe types whose members exhibit behavior. Chapters 6 and 10 described this usage in more detail. Still, *object_classes* are not functions.

A variation of *entity_class* could be defined by:

```
object_class protected_entity_class (entity_class)
  "An entity class which keeps information about who has authorization to
  access its members."
  [authorization_list :    list_of(many, integer);]
```

which adds the capability of using an integer authorization code to allow access to members of that class.

The metaclass definitions of *entity* and *object* are:

```
object_class entity () persistent
  [attributes : class_attributes (attribute);]

object_class object ()
  [attributes : class_attributes (attribute);]
```

12.2 Function Classes

Function classes define types of functions, including the domain and codomain for functions of that class. Function classes are hierarchical. Function subclasses of a function class must have domains which are subtypes of the superclasses's domain.

To clarify the similarities and differences between object classes and function classes, let's look at IDEF1 metaclass definitions. Starting with object types, *entity_class* is an *object_class*, *Employee* is an *entity_class*, and the information kept about "Joe" is an instance of *Employee*. Information about "Joe" is stored in the database.

Now for function classes. *Link_class* is a function class. *Employee_of* from Figure 1.2 is an instance of *link_class*. *Employee_of* is intensional, but the instances of *employee_of* are pairs of entities which are not actually stored by the information system, but rather derived when needed. Thus, whereas information about "Joe" is kept in the database, information about instances of *employee_of* is not.

All in all, it is important to keep in mind that *function_classes* are metaclasses just like *object_classes*. They are classes of classes.

To introduce function classes, we will define a *function_class* called *employee_predicate* which takes an *Employee* entity as its argument and returns true or false. We will start by defining a more general *function_class* called *entity_predicate*. We do this to show the inheritance of function classes. The *function_class* definition is:

```
function_class entity_predicate () (instance_of(entity_class)) : boolean;
```

This declares that *entity_predicate* does not inherit from other *function_classes* (except *function*) and that *entity_predicate* takes one argument which has as its type an instance of the *object_class*. *entity_class*. In other words, the predicate is passed an entity of an *entity_class*, not an *entity_class*. For example, the information kept about an employee is an entity, "Employee" is an *entity_class*.

The definition of *employee_predicate* is:

```
function_class employee_predicate (entity_predicate) (Employee) : boolean;
```

Function classes which inherit from other function classes can restrict the domain of the parent function class. In this case, *employee_predicate* restricts the domain of *entity_predicate* from an entity of any *entity_class* down to an *Employee* entity. We can now define the predicate:

```
employee_predicate older_than_E21_people? (Employee e) : boolean
  "Is the employee older than any member of department E21?"
  [age(e) > (select max(age) from Employee
    where dept = "E21";)]
```

The predicate *older_than_E21_people?* checks to see if the employee is older than any member of department "E21." At compile time the predicate is checked to make sure it conforms with the *employee_predicate* definition. Also, *older_than_E21_people?* can be passed to any function which requires either an *entity_predicate* or an *employee_predicate*.

There are also times when it is desirable that the body of the function be generated automatically. This is almost always the case with *function_classes* for use with methods. An expert (or experts) defines the *function_class*, and then users use the *define* form to supply the necessary parameters to generate the function. This is just a type of macro.

Let's look at an IDEF1 *function_class* which automatically generates instances using the *define* form. Link classes in IDEF1 are functions which map an entity in one entity class to an entity in another entity class. To define a link class, we need to know the independent and dependent entity classes and the name of the link class. In ISyCL, the name of the link class reads from dependent to independent since that direction describes a proper function. Optionally, the independent to dependent name can be given to define the function which maps one independent entity to possibly many dependent entities.

The parameter to the *link_class* function is an entity of the dependent entity class. The function type definition for *link_class* is:

```
function_class link_class () (instance_of(entity_class)) : instance_of(entity_class)
  (symbol name; entity_class DEP; entity_class IND; &optional symbol (inverse ""))
  "Maps an instance of an entity class to an instance of another entity class."
  []
  [link_class ~name (~DEP dep_ent) : ~IND
    [loop for ind_ent in ~IND
      if (for_all inh_attr in ~(inherited_attributes(DEP))
        where (domain@owned_attr(inh_attr) = ~IND)
          (owned_attr(inh_attr)(ind_ent) = apply(inh_attr, dep_ent)))
        return(c);
      end_loop]]
```

This definition certainly requires some explanation. Starting at the top, *link_class* does not inherit from any other function classes. It takes an entity of the dependent entity class and returns an entity from the independent entity class. The parameters of the define form are given in the next line. *Link_class* requires the name of the link class, the dependent entity class, the independent entity class, and, optionally, the name of the function which maps from independent to dependent.

The next line is the documentation string. If *link_class* had attributes, they would be declared in the next block. Finally, there is the macro block which generates the function. See Section 9.3 for a discussion of macro syntax. Most important, remember that the tilde (“~”) causes the next symbol to be evaluated. If the tilde is followed by a form in parentheses, the whole form is evaluated.

To find the independent entity which corresponds with the dependent entity, each of the entities of the independent entity class is checked. The inherited attribute classes of the dependent entity class which were inherited from the independent entity class are checked to see if they have the same values in the independent and dependent entities. If so, there is a mapping from the dependent to independent entities, and that entity of the independent entity class is returned.

To get a better understanding of how this function works, let’s look at an example from Chapter 4 on page 31. First, we define *weak_many_link_class*:

```
function_class weak_many_link_class (link_class) (instance_of(entity_class)) :
                                                    instance_of(entity_class)
0
[cardinality :    string, init_value “weak”];]
```

Weak_many_link_class has a *cardinality* attribute which describes the mapping between entities. The declaration:

```
define weak_many_link_class (employee_of, Emp_Assign, Employee, (inverse assigned_to));
```

generates the function:

```
weak_many_link_class employee_of (Emp_Assign dep_ent) : Employee
[loop for ind_ent in Employee
  if (for_all inh_attr in ‘(EAE#, EAT#, EAP#)
    where (domain@owned_attr(inh_attr) = ‘Employee)
    (owned_attr(inh_attr)(ind_ent) = apply(inh_attr, dep_ent)))
    return(c);
end_loop]
```

This function checks all *Employee* entities to find one whose value of *E#* is the same as the given *Emp_Assign*’s *EAE#* value. When it does find a match, it returns the *Employee* entity.

The only attribute class of *Emp_Assign* inherited from *Employee* is *EAE#*. *EAE#* originates from *Employee*'s *E#* attribute class. Thus, in the *for_all* of the condition of the *if*, only *EAE#* satisfies the *where* condition since *owned_attr(EAE#)* is *E#* and *domain(E#)* is *Employee*. At that point, the value of the *EAE#* attribute of the *Emp_Assign* entity is checked against the value of the *E#* attribute of the *Employee* entity. If the values are equal, the *Employee* entity is returned.

There is still more work to do. The inverse function needs to be defined which maps from independent entities to dependent entities. To accomplish this task, we start by placing a before trigger on define of *link_class*. Thus, whenever a link class is defined, the inverse accessor will be defined if requested. The trigger definition is:

```
before define (link_class lc; symbol name; entity_class DEP;
               entity_class IND; &optional symbol (inverse ""))
[if (not inverse = "")
  gen_inverse_link_class(inverse, IND, DEP);]
```

Like all before triggers, this one takes the same arguments as the function it constrains. If no name is specified for the inverse function, then none is defined. If a name is supplied, the following macro is expanded:

```
macro gen_inverse_link_class (symbol name; entity_class IND; entity_class DEP) : function
[function ~name (~IND ind_ent) : list_of(many, ~DEP)
  [loop for dep_ent of ~DEP
    if (for_all inh_attr in ~(inherited_attributes(DEP))
      where (domain@owned_attr(inh_attr) = ~IND)
      (owned_attr(inh_attr)(ind_ent) = apply(inh_attr, dep_ent)))
    collect dep_ent;
  end_loop]]
```

The function generated is similar to the *link_class* function, other than that this function returns a list of entities. The inverse function *assigned_to* would look like:

```
function assigned_to (Employee ind_ent) : list_of(many, Emp_Assign)
[loop for dep_ent of Emp_Assign
  if (for_all inh_attr in '(EAE#, EAT#, EAP#)
    where (domain@owned_attr(inh_attr) = 'Employee)
    (owned_attr(inh_attr)(ind_ent) = apply(inh_attr, dep_ent)))
    collect dep_ent;
  end_loop]
```

Now the one *define* form generates two functions which map between entities of the entity classes. Best of all, the user need not be concerned with how to write those functions.

The general syntax for *function_class* definitions is:

```
function_class symbol ( <symbol-list> ) ( <symbol-list> ) : <symbol-list>  
  <parm-list>  
  [<doc-string>]  
  [<class-attr-block>  
  <block>]
```

ISyCL Grammar

The following is an attempt to list the grammar for ISyCL. This grammar contains known errors.

A.1 Tokens

<i>letter</i>	: A..Z a..z
<i>digit</i>	: 0..9
<i>hexdigit</i>	: 0..9 A..F a..f
<i>char</i>	: <i>letter</i> <i>digit</i> # \$? _
<i>char-spec</i>	: # \ [<i>char</i> <i>hexdigit hexdigit</i>]
<i>whitespace</i>	: \space \tab \newline
<i>identifier</i>	: [%] <i>letter character</i> *
<i>string</i>	: "[<i>digit</i> <i>character</i> <i>whitespace</i>]"
<i>int</i>	: <i>digit</i> ⁺
<i>posint</i>	: + <i>digit</i> ⁺
<i>negint</i>	: - <i>digit</i> ⁺
<i>intnum</i>	: <i>int</i> <i>posint</i> <i>negint</i>
<i>realnum</i>	: <i>intnum</i> . <i>int</i> [(<i>E</i> / <i>e</i>) <i>intnum</i>]

A.2 Productions

This section is the BNF for ISyCL. Certain conventions are used in presenting the BNF. Words in *italics* refer to tokens defined in the section above. **Bold** symbols represent those tokens that must appear in the statement for the statement to be valid. There is a bit of semantics thrown in to the BNF. *Type*, *function*, *subrange*, *class*, *object_class*, *function_class*, and *metaclass* refer to *identifiers* which are either built-in or have been defined using meta-layer constructs.

A.2.1 Lists of Tokens

<code><int-list></code>	<code>::=</code>	<code>int</code> <code> </code> <code>int , <int-list></code>
<code><identifier-list></code>	<code>::=</code>	<code>identifier</code> <code> </code> <code>identifier, <identifier-list></code>

A.2.2 Definitions

<code><function-call></code>	<code>::=</code>	<code><accessor> (<expression-list>)</code>
<code><accessor-list></code>	<code>::=</code>	<code><accessor></code> <code> </code> <code><accessor> , <accessor-list></code>
<code><accessor></code>	<code>::=</code>	<code>[~]identifier</code> <code> </code> <code>identifier [<identifier-list>]</code>
<code><constant-list></code>	<code>::=</code>	<code><constant></code> <code> </code> <code><constant> , <constant-list></code>
<code><constant></code>	<code>::=</code>	<code><char-spec></code> <code> </code> <code><num-constant></code> <code> </code> <code><boolean-constant></code> <code> </code> <code>string</code>
<code><doc-string></code>	<code>::=</code>	<code>string</code>
<code><boolean-constant></code>	<code>::=</code>	<code>true</code> <code> </code> <code>false</code>
<code><num-constant></code>	<code>::=</code>	<code>intnum</code> <code> </code> <code>realnum</code>

A.2.3 Types and Classes

<code><class-list></code>	<code>::=</code>	<code><class></code> <code> </code> <code><class> , <class-list></code>
<code><class></code>	<code>::=</code>	<code><type></code> <code> </code> <code>class</code> <code> </code> <code>metaclass : class</code> <code> </code> <code>model :: metaclass :class</code>
<code><type-list></code>	<code>::=</code>	<code><type></code> <code> </code> <code><type> , <type-list></code>
<code><type></code>	<code>::=</code>	<code><base-type></code> <code> </code> <code><function-call></code>

<base-type>	::=	array
		boolean
		character
		class
		fixed
		float
		function
		integer
		<i>subrange</i>
		symbol
		type

A.2.4 Relational Operators

<relation-op>	::=	<rel-op>
		<equiv-op>
		<memb>
<equiv-op>	::=	= == <>
<memb>	::=	in of of_type
<rel-op>	::=	< > <= >=

A.2.5 Boolean Expressions

<condition>	::=	(<boolean-expr>)
<boolean-expr>	::=	<boolean-conj> <bool-expr>
<bool-expr>	::=	[!] iff <bool-expr>
		ϵ
<boolean-conj>	::=	<boolean-disj> <bool-conj>
<bool-conj>	::=	[!] { or xor } <bool-conj>
		ϵ
<boolean-disj>	::=	<boolean-inv> <bool-disj>
<bool-disj>	::=	[!] and <bool-disj>
		ϵ
<boolean-inv>	::=	[!] not <boolean-primary>
		<boolean-primary>
<qualified-expr>	::=	<for-all> <for-some>
<for-all>	::=	for_all <identifier-list> <qualifier>
		[<where-clause>] [<on-failure>] <condition>
<for-some>	::=	for_some <identifier-list> <qualifier>
		[<where-clause>] [<on-failure>] <condition>
<where-clause>	::=	where <condition>

<on-failure>	::=	on_failure <block>
<qualifier>	::=	<in-clause> of <class>
<in-clause>	::=	in { <set-access> <seq-access> }
<set-access>	::=	'{ <expression-list> } <set-builder> <common-access>
<set-builder>	::=	{ <accessor-list> <qualifier> <boolean-expression> }
<seq-access>	::=	<bag-access> <list-access> <ordered-set-access> <string-access>
<bag-access>	::=	<common-access>
<list-access>	::=	'(<expression-list>) <common-access>
<ordered-set-access>	::=	<common-access>
<string-access>	::=	<i>string</i> <accessor> <function-call>
<common-access>	::=	<select> <accessor> <function-call>
<logical-if>	::=	<condition> -> <boolean-expr>
<boolean-primary>	::=	<predicate> [~](<boolean-expr>)
<predicate>	::=	<boolean-statement> <expression> <prd-subexpr> <boolean-constant> <function-call> <accessor>
<pred-subexpr>	::=	<relation-op> <expression> <rel-op> <expression> <rel-op> <expression>

A.2.6 Expressions

<code><expression></code>	<code>::=</code>	<code><term> <expr></code>
<code><expr></code>	<code>::=</code>	<code>[!] { + - } <expr></code> <code>ε</code>
<code><term></code>	<code>::=</code>	<code><factor> <term2></code>
<code><term2></code>	<code>::=</code>	<code>[!] { * / mod } <term2></code> <code>ε</code>
<code><factor></code>	<code>::=</code>	<code><primary> [!][^] { <i>intnum</i> <i>realnum</i> }</code> <code><primary></code>
<code><primary></code>	<code>::=</code>	<code><constant></code> <code><accessor></code> <code><function-call></code> <code><set-builder></code> <code><select></code> <code><explicit-list></code> <code>[~](<expression>)</code>
<code><expression-list></code>	<code>::=</code>	<code><expression> , <expression-list></code> <code><expression></code>

A.2.7 Declarations

<code><type-decl></code>	<code>::=</code>	<code><accessor-list> : <class></code> <code>[, <i>init_value</i> <expression>]</code>
<code><parm-list></code>	<code>::=</code>	<code>(<parm-decl-list>)</code> <code>()</code>
<code><parm-decl-list></code>	<code>::=</code>	<code><parm-decl></code> <code><parm-decl> ; <parm-decl-list></code>
<code><parm-decl></code>	<code>::=</code>	<code><class> <accessor-list></code> <code><parm-keyword></code>
<code><parm-keyword></code>	<code>::=</code>	<code>&quote { identifier () {} }</code> <code><accessor-list></code> <code>&rest class identifier</code>

A.2.8 Statements

<code><statement></code>	<code>::=</code>	<code><assignment></code> <code><if></code> <code><case></code> <code><choose></code> <code><loop></code> <code><with></code>
--------------------------------	------------------	--

<statement-list>	::=	<statement> <statement-list> <statement>
<assignment>	::=	<lhs> := <rhs> ;
<if>	::=	if <condition> <action> [else <action>]
<case>	::=	case <expression> [testing_with <relation-op>] <expr-action-list> end_case
<expr-action>	::=	<expression> : <action>
<expr-action-list>	::=	<expr-action> <expr-action> <expr-action-list>
<choose>	::=	choose [first <boolean-constant> until <boolean-constant> all <boolean-constant>] <cond-action-list> end_choose
<cond-action>	::=	<condition> : <action>
<cond-action-list>	::=	<cond-action> <cond-action> <cond-action-list>
<select>	::=	select [all distinct] <selection> <table-expr> end_select
<with>	::=	with <decl-list> <block>
<decl>	::=	<type-decl> <exception> <trigger>
<decl-list>	::=	<decl> <decl> <decl-list> ε
<exception>	::=	exception identifier <parm-list> continue end <action>
<insert>	::=	insert into class (<identifier-list>) values <list-access-list> ;
<list-access-list>	::=	<list-access> <list-access> , <list-access-list>
<cursor>	::=	declare identifier cursor_for <select> ;
<lhs>	::=	<accessor> <accessor-list>

<rhs>	::=	<expression> <boolean-expression>
<selection>	::=	<expression-list> *
<table-expr>	::=	<from-clause> [<where-clause> <order-by-clause> <having-clause>]
<from-clause>	::=	from <table-ref-list>
<order-by-clause>	::=	order_by { asc desc } [<identifier>]
<having-clause>	::=	having <predicate>
<table-ref-list>	::=	<table-ref> <table-ref> , <table-ref-list>
<table-ref>	::=	<table> [<range-variable>]
<loop>	::=	loop {<iteration-list> {<control-list>} {<main-list>} {<control-list>}} end_loop
<iteration>	::=	<for-clause> <i>integer times</i>
<iteration-list>	::=	<iteration> <iteration-list> <iteration> ϵ
<control>	::=	<termination> <initial-final>
<control-list>	::=	<control> <control-list> <control> ϵ
<main>	::=	<accumulation> <statement> <loop-finish>
<main-list>	::=	<main> <main-list> <main> ϵ
<termination>	::=	<until> <while>
<initial-final>	::=	<initially> <finally>

<accumulation>	::=	<collect> <append> <count> <sum> <maximize> <minimize>
<initially>	::=	initially <action>
<finally>	::=	finally <action>
<collect>	::=	collect { <internal> <into-var> }
<append>	::=	append { <internal> <into-var> }
<count>	::=	count { <internal> <into-var> }
<sum>	::=	sum { <internal> <into-var> }
<maximize>	::=	maximize { <internal> <into-var> }
<minimize>	::=	minimize { <internal> <into-var> }
<internal>	::=	<type-spec> <expr> <var>
<into-var>	::=	<expr> into <var>
<for-clause>	::=	for <for-subclause> [and <for-subclause>]
<for-subclause>	::=	<for-arithmetic> <for-in-list> <for-on-list> <for-equals-then> <for-across>
<for-arithmetic>	::=	<var> [{ from downfrom upfrom } <expr>] [{ to downto upto below above } <expr>] [by <expr>]
<for-in-list>	::=	<var> in <expr> [by <i>step-func</i>]
<for-on-list>	::=	<var> on <expr> [by <i>step-func</i>]
<for-equals-then>	::=	<var> := <expr> [then <i>step-func</i>]
<for-across>	::=	<var> across <i>array</i>

A.2.9 Units

<action>	::=	<statement> <block>
<block>	::=	[<statement-list>]
<func-block>	::=	<decl-list> <block>

<function>	::=	{ function <i>function-class</i> } <i>identifier</i> { <func-assign> <func-def> }
<func-assign>	::=	:= <function-comp> ;
<func-def>	::=	<parm-list> [: <identifier-list>] [<doc-string>] [(<attr-assign-list>)] <func-block>
<attr-assign-list>	::=	<assignment> <assignment> <attr-assignment-list>
<macro>	::=	macro <i>identifier</i> <identifier-list> [<doc-string>] <block>
<define>	::=	define <i>identifier</i> (<expression-list>)
<trigger>	::=	<before-after> <during>
<before-after>	::=	{ before after } <i>identifier</i> of [<attrib-of>] <i>class</i> <i>identifier</i> <parm-list> [<doc-string>] <action>
<during>	::=	during <i>identifier</i> of [<attrib-of>] <i>class</i> <i>identifier</i> <parm-list> [<doc-string>] <decl-list> <block>
<attrib-of>	::=	attrib <identifier-list> of
<class-def>	::=	{ class <i>object-class</i> } <i>identifier</i> (<identifier-list>) [<doc-string>] <CAB-list> [(<key-list>)]
<class-attr-block>	::=	[[<required-attr-list>] <attr-decl-list>]
<CAB-list>	::=	<class-attr-block> <CAB-list> <class-attr-block> ε
<required-attr-list>	::=	(<req-attr-decl-list>)
<attr-decl>	::=	<i>identifier</i> : <type> [, <attr-option-list>] ;
<attr-decl-list>	::=	<attr-decl> <attr-decl-list> <attr-decl> ε
<req-attr-decl-list>	::=	<attr-decl> <attr-decl-list> <attr-decl>
<attr-option-list>	::=	<attr-option> <attr-option> , <attr-option-list>

<attr-option>	::=	inverse <i>identifier</i> read_only non_null init_value <expression>
<key-list>	::=	<key> <key> <key-list>
<key>	::=	(<identifier-list>)
<type-def>	::=	type <i>identifier</i> := <type>
<insert>	::=	insert [<i>identifier</i>] into <class-access> (<accessor-list>) values (<expression-list>)
<delete>	::=	delete <select-from> [<where-clause>]

A.2.10 Meta Units

<subrange>	::=	subrange <accessor> (<type> <accessor>) [<doc-string>] <condition>
<function-class>	::=	function_class <i>identifier</i> (<identifier-list>) (<identifier-list>) : <identifier-list> <parm-list> [<doc-string>] [<class-attr-block> <block>]
<object-class>	::=	object_class <i>identifier</i> (<identifier-list>) [<obj-class-keywords>] [<doc-string>] <attribute-block>
<obj-class-keywords>	::=	<obj-class-keyword> <obj-class-keyword> , <obj-class-keywords>
<obj-class-keyword>	::=	persistent no_inheritance no_key
<attribute-block>	::=	[<attr-list>]
<attr-list>	::=	<class-attr-decl-list> <attr-decl-list>
<class-attr-decl>	::=	<i>identifier</i> : class_attributes (<i>identifier</i>) ;
<class-attr-decl-list>	::=	<class-attr-decl> <class-attr-decl-list> <class-attr-decl> ε

Revision Notes

Beta Draft 1.0 (October 23, 1989)

Initial publicly distributed draft. Beta Draft 1.0 was distributed at the IDEF Users' Group Meeting on October 24, 1989. Purpose of the distribution was to generate comments from the community.

Beta Draft 1.1 (November 2, 1989)

IISyCL levels are now called layers. The IISyCL layers are:

- Area Expert Layer,
- Analyst Layer,
- Information Systems Design (Systems) Layer,
- Method Formalization (Meta) Layer.

The "IISyCL Grammar" and "IDEF1 Metatypes" appendices have been removed. The grammar will return to the document when it is proven to be complete and correct. The IDEF1 metatypes will resurface as part of a new technical report.

Various wording problems have been corrected.

Beta Draft 1.2 (December 22, 1989)

Many changes and additions have been made, but the changes are minor enough not to warrant a major version change.

The *bag* type has been added to round-out the "set" types. See Sections 2.3.2 and 6.2.2.

Explicit lists can now be created using the list function instead of just parentheses. Just parentheses caused confusion between "grouping" and "list construction" (see Section 6.2.6.1).

The quantifier and loop syntaxes have been extended to facilitate sets of pairs. For example,

```
loop for (e,d) in get_set_of_emp_dept_pairs()
  if (dept(e) = d)
    count e into total;
end_loop
```

determines the number of pairs in the set generated by *get_emp_dept_pairs* in which the employee works in the department.

The type *type* has been removed. *Type* was originally the type of all types. It is necessary to distinguish between metatypes. For instance, *array* is of type *object_type*, whereas *Employee* might be of type *entity_class*. They are not both of type *type*. User-defined metatypes will nearly always be based on *object_type* or *function_type*.

Numerous wording changes and error corrections have been made.

Beta Draft 2.0 (March 6, 1990)

As denoted by the jump in version number to 2.0, many critical changes have been made to ISyCL and this report. First, the name is now the Information Systems Constraint Language (ISyCL) instead of Integrated Information Systems Constraint Language. There is no reason why ISyCL could not be used for any type of information system.

The most important change to the language is the addition of classes. In the beginning, ISyCL had both classes and types. Prior to release of the first version of this report, the distinction between class and type was removed in order to bring more uniformity to the language. Upon review, it has been decided that the distinction is necessary, and thus classes are back.

Why is there a need for both classes and types? Types describe data structures which are understood by the ISyCL processor. Classes describe higher level types which are defined by the user. Instances of types can be stored as attribute values of entity classes (database objects), whereas object classes cannot. There are classes of the same name associated with each type, but new classes cannot inherit from these built-in classes. This inability to inherit from types is one of the primary reasons for drawing the distinction between classes and types.

Another reason for the split is the usage of the terms “type” and “class” in other languages. ISyCL should now be more intuitive to people who know the Common LISP Object System or C++. Also, IDEF4 (object-oriented design method) and ISyCL now fit together better.

The type *type* is back. All built-in types are of type *type*.

The introduction has been elaborated to more fully describe the purpose for developing ISyCL.

ISyCL symbols are now only case-sensitive if there is an “%” in front of them. This allows functions with case-sensitive names from other languages to be called directly, while relieving the user from the burden of remembering the case of symbols. All symbols not preceded by a “%” are converted to uppercase by the ISyCL processor.

A more comprehensive syntax for explicit list and set structures has been added. Structures which have order (lists and ordered sets) are declared using parentheses, whereas unordered groupings (sets and bags) are declared using braces. Groupings which allow duplicates are preceded by a quote ('). Groupings which do not allow duplicates are preceded by a backquote (`). Also, members of explicit groupings are not evaluated unless preceded by a tilde (~). For example:

```
'(1, a, ~c)
```

would generate a bag containing the integer “1,” the symbol *a*, and the value of *c*.

List_of is now the only built-in function for generating subtypes of list. See Section 6.2.6.1 for more details.

Membership is now specified by *in*, *of*, or *of_type*. *In* shows membership in a sequence or set, *of* shows membership in a class, and *of_type* shows membership in a type. Remember that all types are also classes.

Minor syntactic changes were made to case and choose to get rid of “=>” which some people confused with “>=.”

Finally, the grammar list has returned. If there are any discrepancies between the grammar in the text and the grammar in the appendix, the appendix should be considered correct.

Beta Draft 2.1 (June 30, 1990)

This report is now called the “ISyCL Technical Report.” Wording and oversight changes have been made. The grammar list is still incomplete.

Final Report (March 8, 1991)

This is the final report on ISyCL under the Integrated Information Systems Evolution Environment (IISSE) project. This version is very different from the others in both form and content. A great deal of effort has gone into trying to ensure the contents of this report accurately reflect the current state of the language.

The first section of the report provides an overview of the language which should be sufficient for anyone who just wants to know what ISyCL is. The language reference section provides a more detailed, yet readable, description of ISyCL.

References

- ANSI/X3/SPARC, Study Group on Data Base Management Systems, Interim Report, 75-02-08, 1975.
- D. Appleton Company, "IISS - Integrated Information Support System", ICAM Project Priority 6201, Subcontract #013-078846, USAF Prime Contract #F33615-80-C-5155, December 31, 1985.
- ISO, Concepts and Terminology for the Conceptual Schema and Information Base, edited by J. J. van Griethuysen, March 15, 1982.
- Mayer, R. J., "IDEF1 - Information Modeling; Theory and Practice", Department of Industrial Engineering, Texas A&M University, 1987.
- Mayer, R. J. et al., "Analysis of Methods," Knowledge Based Systems Laboratory Technical Report (KBSL-89-1001), 1989.
- Menzel, C. P. and R. J. Mayer, "Theoretical Foundations for Information Representation and Constraint Specification," Knowledge Based Systems Laboratory Technical Report (KBSL-89-1005), 1989.
- PDES Form Feature Information Model, Version 4 (Draft), August 17, 1988.
- SofTech, "Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual (IDEF0)," Technical Report UM 110231100, June 1981.
- Wells, M. S. and R. J. Mayer, "IDSE User's Manual — Version 1," Knowledge Based Systems Laboratory Technical Report, 1988.
- Zachman, J. A., "A Framework for Information Systems Architecture," IBM Report No. G320-2785, IBM Los Angeles Scientific Center, 1986.